Linuxプログラミング

例題で学ぶUNIXプログラミング環境のすべて

Neil Matthew & Richard Stones 著
葛西重夫 訳

wrox
SOFT BANK Publishing

Linuxプログラミング

例題で学ぶUNIXプログラミング環境のすべて

Neil Matthew & Richard Stones 著　葛西重夫 訳

Beginning Linux Programming

By Neil Matthew & Richard Stones

Japanese language edition published by SOFTBANK CORP.

Japanese translation rights arranged with Wrox Press
through Japan UNI Agency, Inc., Tokyo.

はじめに

本書は、『Beginning Linux Programming』（Neil Matthew & Richard Stones著、Wrox Press刊）を翻訳したものです。原著はLinux Journal誌などにもたびたび取り上げられ、Linux/UNIXプログラミングの学習書、解説書として、欧米では高い評価を得ています。

本書の主な特徴は次のとおりです。

- **Linuxを通じてUNIXプログラミングの多彩な技法を学ぶことができる**

  UNIXプログラミングの世界は、独自の伝統と哲学を持っています。本書はLinuxという比較的新しい実装を取り上げながら、この実装に固有のトピックだけに絞り込むことをあえてせず、Linuxを生み出すに至ったUNIXプログラミングの世界を尊重しつつ、そこでつちかわれてきたさまざまな技法を、この新しい実装を通じて学ぶことができるように配慮しています。

- **サンプルプログラムを発展させながら実践的な技法を身につけることができる**

  本書では、各機能の用法を示すサンプルコードを豊富に提供するとともに、全体を通じてひとつの実用的なアプリケーションを用意し、そのつど説明している技法を活用してプログラムを作り直しながら、このアプリケーションを発展させていきます。読者は同じアプリケーションを何度も設計し直し、実装し直すことによって、実践的な技法を確実に身につけることができます。

翻訳にあたっては、原著の記述とサンプルコードを全面的に見直し、適宜修正、加筆するとともに、最新のシステムで実証することを心がけました。なお、原著の構成中、一部付随的と思われる箇所は訳出していません。

本書の対象読者

本書は、読者がUNIXに関する基本的な知識をすでに持っていること、さらにCまたはC++によるプログラミングの経験を多少なりとも持っていることを前提としています。なお、プログラミングの経験はUNIXでのプログラミングである必要はなく、MS-DOSまたはMicrosoft Windowsでのプログラミングであってもかまいません。

本書の表記

本文中、関数、構造体、コマンドなどの構文は、次のように枠で囲んで示します。

```
#include <stdio.h>

int printf (const char *format, ...);
```

また、コマンドラインでの入力は太字で示し、出力は細字で示します。

```
$ grep "command line" introduction
When the command line is shown,
it's in the above style,
whereas output is in this style.
```

コードを示す場合、新しく登場するコードや重要なコード、あるいはその場の説明と関係の深いコードは、太字と網かけで強調表示します。一方、すでに登場したコードや重要度の低いコード、その場の説明とあまり関係のないコードは細字で示します。

```
Lastly in our code examples,
the code foreground style shows
new, important, pertinent code;
while code background shows
code that's less important in the present context,
or has been seen before.
```

サンプルコードについて

本書で取り上げるサンプルプログラムのフルソースコードは、次のサイトからダウンロードできます。

```
http://www.softbank.co.jp/books/editor/1st/linux/
```

注意

本書では翻訳にあたって、Slackware、Red Hat、Debian、TurboLinuxなど、主だったLinuxディストリビューションでサンプルプログラムの動作を検証しました。基本的に動作することが確認されていますが、環境によっては問題が発生する可能性があります。詳細については、ソースコードに付属するreadmeファイルを参照してください。

メモ

本書では、サンプルプログラムをあえて日本語化していません。これは、日本語環境が構築されていないシステムでも実行できるようにしておくこと、およびプログラムが全世界で通用するようにしておくことを配慮した結果です。

原著に含まれるサンプルプログラムのフルソースコードは、次のサイトからダウンロードできます。

```
http://www.wrox.com/
```

本書、およびその原著のサンプルプログラムは、GNU General Public License（GPL）の条件のもとで自由に改変、配布することができます。GPLについては、付録Bに全文を示してあります。

目次

第1章

第2章

第5章

第6章

第11章

プロセス間通信とパイプ 437

第12章

第13章

付録A

付録B

付録C

第1章

さあ、はじめよう

第1章では、Linuxとはなにか、Linuxの源流となったUNIXとLinuxの間にはどのような関係があるかについて説明します。また、UNIXの開発システムで利用できる各種のツールや機能についてざっと紹介し、簡単なプログラムを作成して実行します。この章で取り上げる主な項目は次のとおりです。

- UNIX、Linux、GNU
- UNIXのプログラムとプログラミング言語
- 開発ツールや開発リソースの場所
- 静的ライブラリと共有ライブラリ
- UNIXの哲学

1.1 UNIXとは

UNIXオペレーティングシステムは、電気通信分野の巨人AT&Tの一部門であるベル研究所で開発されました。1970年代にDECのPDPコンピュータ用に設計されたUNIXオペレーティングシステムは、現在ではPCワークステーションからマルチプロセッササーバー、さらにスーパーコンピュータにいたる多様なハードウェアプラットフォームで、マルチユーザー／マルチタスクのオペレーティングシステムとして広く利用されています。

厳密にいうと、UNIXは**X/Open**が管理する商標であり、X/Open仕様XPG4.2に準拠したコンピュータオペレーティングシステムのことを指します（注1）。X/Open仕様XPG4.2はSPEC1170とも呼ばれ、UNIXオペレーティングシステムの全関数の名前、インタフェース、動作を定義しています。このX/Open仕様は、それ以前の一連の仕様、およびIEEEによって積極的に開発が進められてきたP1003（**POSIX**仕様）のスーパーセットになっています（注2）。

現在では、SunのSparc用またはIntel用のSolarisなど、商用のものから、FreeBSDやLinuxといったフリーのものまで、数多くのUNIXライクなシステムが利用できます。ただし、これらのシステムのうちX/Open仕様に準拠しているものはそれほど多くありません。これは、X/Open仕様がごく最近になって発表されたものであることが原因です。POSIXが果たした役割はけっして小さくありませんでしたが、これまでさまざまなUNIXシステム相互間で互換性の問題があったことも事実です。しかし、X/Open仕様の発表によって、今後UNIXとその他の多くのUNIX系システムが同じ方向を目指して歩んでいくと期待されます。

注1　X/Openは、UNIXシステムの標準仕様を定めるために設立された業界団体。その成果は、XPG（X/Open Portability Guide）として公開されている。1996年、X/OpenはOSF（Open Software Foundation）と合併し、The Open Groupとなった。

注2　POSIX（Portable Operating System Interface for UNIX）は、IEEE（Institute of Electrical and Electronic Engineers：米国電子電気学会）によって策定された、UNIXシステムの標準インタフェース規格。

1.2 Linuxとは

Linuxは、オペレーティングシステムの低レベルのコアとなるUNIXライクなカーネルを自由に配布できる形で実装したものです。LinuxはUNIXシステムに触発されて生まれたものなので、LinuxとUNIXのプログラムは互いによく似ています。実際、UNIX用に記述されたほとんどのプログラムは、Linux上でコンパイルして実行することができます。また、商用UNIX向けの市販アプリケーションには、バイナリ形式のまま手を加えることなくLinux上で実行できるものもあります。Linuxを開発したのは、ヘルシンキ大学のLinus Torvaldsです。開発には、インターネット上の多くのUNIXプログラマも協力しました。Linuxは、小規模なUNIXシステムであるAndrew TanenbaumのMINIXをもとに、一種の趣味的な企てとして開発が始まり、やがてそれ自体で完結した完全なUNIXシステムへと成長しました。Linuxカーネルは、AT&Tのソースやその他のベンダー固有のソースコードをいっさい使用していません。

1.2.1 ディストリビューション

上に述べたように、Linuxは実際にはカーネルの部分だけを指します。カーネルのソースを入手し、コンパイルしてインストールしたあと、フリーで配布されているその他の多くのソフトウェアプログラムをインストールすれば、完全なUNIXライクなシステムを構築できます。こうしてインストールされたさまざまなシステムは、カーネル以外にも多くのものを含んでいますが、一般に"**Linuxシステム**"と呼ばれています。ユーティリティのほとんどには、Free Software FoundationのGNUプロジェクト（後述）のものが使われています。

しかし、ソースコードだけを出発点にLinuxシステムを作成することは必ずしも容易ではありません。さいわいなことに、カーネルのほかにもさまざまなプログラミングツールやユーティリティを収めた**ディストリビューション**が多くの人々の手によって作成され、CD-ROMなどで提供されています。通常、これらのディストリビューションには、各種UNIXシステムに共通のグラフィカル環境であるX Window Systemに加えて、Linuxシステムのインストールを支援するためのセットアッププログラムや、有益な情報を含んだテキストファイルなどが付属しており、こうしたすべてのものがCD-ROMに収められています。よく使われているディストリビューションにSlackware、Debian、Red Hatなどがありますが、このほかにも数多くのディストリビューションが存在します。

1.2.2 GNUプロジェクトとFree Software Foundation

Linuxは、たくさんの人々の共同作業によって支えられています。オペレーティングシステムのカーネルそのものは、システムのほんの一部分を占めるにすぎません。現在では商用UNIXシステムでも、システムサービスを提供するアプリケーションプログラムや各種ツールをあらかじめ組み込んで出荷するのが一般的になっています。Linuxシステムの場合、この種のプログラムには、多くのプログラマが作成し、無償で公開しているものを使用しています。

Linuxコミュニティは、ほかのさまざまなコミュニティとともにフリーソフトウェアという考え方を支持しています。フリーソフトウェアとは、**GNU General Public License（GPL）**に基づいて種々の制限を取り除いたソフトウェアのことです。フリーソフトウェアの入手にはいくばくかの費用（通信費など）がかかりますが、入手したソフトウェアは自由に使うことができます。通常、フリーソフトウェアはソースの形で配布されています。

Free Software Foundationは、UNIXをはじめとする各種システム用エディタとして名高いGNU Emacsの作者、Richard Stallmanによって設立されました。Stallmanはフリーソフトウェア理念の提唱者で、UNIX互換のオペレーティングシステムと開発環境の作成を目的とする**GNU**プロジェクトをスタートさせました。GNUプロジェクトが目指しているUNIX互換のオペレーティングシステムと開発環境は、最も低レベルな面ではUNIXと異なる部分を持つ可能性もありますが、UNIXのアプリケーションは問題なくサポートされます。GNUは、“GNU's Not Unix”の略です。

GNUプロジェクトは、UNIXシステムのものとほぼ同じ動作をするアプリケーションをこれまでにもすでに多数、ソフトウェアコミュニティに提供しています。これらのプログラムはGNUソフトウェアと呼ばれ、GNU General Public License（GPL）の条件のもとで配布されています。GPLには、フリーソフトウェアの使用に関して他者が制限を設けることを禁じた“**copyleft**”という考え方についての具体的な説明があります。本書の付録Bにも収録していますので、ぜひ一度読んでみるとよいでしょう。ちなみに、copyleftはcopyrightをもじった造語です。

GPLに従って配布されているソフトウェアには、次のようなものがあります。

- `gcc`　Cコンパイラ
- `g++`　C++コンパイラ
- `gdb`　ソースコードレベルデバッガ
- `gnumake`　UNIX `make`の1バージョン
- `bison`　UNIX `yacc`互換のパーサジェネレータ
- `bash`　コマンドシェル
- `emacs`（`mule`）　テキストエディタと環境

1.3 Linuxでのプログラミング

多くの人は、UNIXでのプログラミングとはすなわちCを使うことだと考えています。たしかに、UNIXはもともとCで記述され、UNIXアプリケーションの大多数もCで記述されています。しかし、UNIXプログラマが利用できる選択肢はCだけにとどまりません。本書では主にCを取り上げますが、Perl、HTML、Java、Tclなど、状況や目的に応じてC以外にもさまざまなプログラミング言語を利用するとこができます。

メモ

厳密にいえば、UNIXの一番最初のバージョンは、1969年にPDP 7アセンブラ言語によって記述されました。Cはこれと同じ頃、Dennis Ritchieによって設計され、1973年にはDennis RitchieとKen ThompsonによってUNIXカーネルがCで書き直されました。これは、システムソフトウェアがアセンブラ言語で記述されていた当時としては画期的なできごとでした。

UNIXシステムでは実にさまざまなプログラミング言語を利用できますが、そのうちの多くは、CD-ROMやインターネット上のFTPサイトなどから自由に手に入れることができます。UNIXプログラマが利用できるプログラミングツールには、次のようなものがあります。

C	C++	Objective C
Fortran	Pascal	Ada
Lisp	Scheme	Forth
Prolog	Icon	Eiffel
Oberon	sh（UNIX Borne Shell）	Tcl/Tk
Python	Perl	Java
PostScript	HTML	Smalltalk
Modula 2	Modula 3	SQL

本書では、このうちUNIXシェルとCに焦点をあてて解説します。まず、次の第2章でUNIXシェル（**sh**）を取り上げ、小～中規模程度のアプリケーションを開発する方法について説明します。第3章以降では、UNIXの各種プログラミングインタフェースについて、Cプログラマの視点から詳しく見ていきます。

1.3.1 UNIXプログラム

UNIXのアプリケーションを大別すると、**実行可能ファイル**と**スクリプト**という2つの特別な種類のファイルに分類できます。実行可能ファイルは、コンピュータが直接実行できるプログラムで、DOSの`.EXE`ファイルに相当します。スクリプトは、一連の命令を書き並べたもので、これをほかのプログラム（インタープリタ）が解釈して実行します。DOSの`.BAT`ファイルやBASICプログラムなどにあたります。

UNIXでは、実行可能ファイルやスクリプトに特別な名前を付けたり、特定の拡張子を追加したりする必要はありません。指定されたファイルが実行可能なプログラムかどうかは、ファイルシステムの属性（第2章で説明します）を見ることで判断するからです。このため、UNIXでは、既存のスクリプトをコンパイル済みのプログラムで置き換えたり、その逆の操作を行ったりしても、該当するスクリプトまたはプログラムを使うほかのプログラムやユーザーには影響を与えずにすみます。実際、ユーザーレベルでは、スクリプトとプログラムの間に本質的な違いはありません。

UNIXシステムにログインしたユーザーは、**シェルプログラム**（通常は`sh`）と対話することになります。シェルプログラムは、DOSの`COMMAND.COM`同様、ユーザーに代わってプログラムを実行する役割を果たします。ユーザーがプログラムを名前で指定すると、シェルプログラムは、指定された名前を持つファイルを一連のディレクトリの中から探します。検索の対象となるディレクトリは、DOSの場合と同じようにシェル変数`PATH`に収められています。この検索パスは、ユーザーが自分で追加することもできますが、ふつうはシステム管理者が設定します。通常、検索パスには、システムのプログラムが格納されている標準的な場所が設定されています。次に示すのは、一般的な検索パスと、各パスに収められているプログラムの種類です。

- `/bin`　システム起動時に使われるバイナリ、プログラム
- `/usr/bin`　ユーザー用のバイナリ、標準プログラム
- `/usr/local/bin`　各システム固有のローカルバイナリ、プログラム

UNIXでは、`PATH`変数の各エントリを区切るのにコロン（`:`）を使います。セミコロン（`;`）を使うDOSとは、この点が異なります。次に、典型的な`PATH`変数の値を示します。

```
/usr/local/bin:/bin:/usr/bin:.:/usr/X11/bin:/usr/openwin/bin
```

1.3.2 Cコンパイラ

なにはともあれ、まずはCを使ったUNIXでの開発を実際に体験してみましょう。ここでは、おなじみの"Hello World"を表示するプログラムを記述し、コンパイルして実行します。

例題 最初のUNIXプログラム

1 ファイル`hello.c`のソースコードは次のとおりです。

```
#include <stdio.h>

int main()
{
    printf("Hello World\n");
    exit(0);
}
```

2 このプログラムは、なんらかのエディタを使って入力する必要があります。標準的なLinuxシステムで使用できるエディタにはさまざまなものがあります。たとえば、多くのユーザーに人気のあるエディタに`vi`があります。筆者は`emacs`を好んで使っています。`emacs`はたいへん強力なエディタなので、この機会に少しかじっておくとよいでしょう。`emacs`の使い方を覚えるには、**Ctrl-H**を押したあと、チュートリアル（tutorial）の頭文字**t**を入力します（`mule`では、**Ctrl-H**を押したあと、大文字の**T**を入力すると、日本語チュートリアルを起動できます）。`emacs`では、マニュアルのすべてをオンラインで参照することができます。**Ctrl-H**を押したあと、**i**を入力してみてください。

3 POSIX準拠のシステムでは、Cコンパイラは`c89`という名前になっています。もともと、Cコンパイラは単に`cc`と呼ばれていました。年とともにさまざまなベンダーがUNIXライクなシステムを販売するようになり、付属するCコンパイラもベンダーごとに異なる機能やオプションを持つようになりましたが、名前だけはもとの`cc`がそのまま使われています。
POSIX規格が作成された際、こうしたさまざまなベンダーのすべてのCコンパイラと互換性のある標準的な`cc`コマンドを定義することは不可能でした。そこで、POSIXでは、Cコンパイラの新しい標準的なコマンドである`c89`を作成することにしたのです。このコマンドが存在する場合には、どのようなマシンで実行するかにかかわらず、常に同じオプションを使うことができます。
Linuxシステムでは、`c89`（存在する場合）、`cc`、`gcc`というコマンドはいずれも、システムのCコンパイラ（通常はGNU Cコンパイラ）を指しています。UNIXシステムでは、Cコンパイラの名前はほとんど常に`cc`になっています。
本書では、さまざまなLinuxディストリビューションに標準で付属していることや、ANSI標準のCの構文をサポートしていることから、GNU Cを使用することにします。GNU Cがイ

ンストールされていないUNIXシステムを使っている場合には、この機会にGNU Cを入手してインストールするとよいでしょう。本書でGNU Cを使う場合には、便宜上**cc**とだけ表記しますが、使っているシステムに応じて適切なコマンド名に置き換えて読んでください。

4 さて、入力したプログラムをコンパイルしてリンクし、実際に実行してみましょう。

```
$ cc -o hello hello.c
$ hello
Hello World
$
```

解説

上の例では、システムのCコンパイラを起動し、Cのソースコードを実行可能ファイル**hello**に変換しました。作成されたプログラムを実行した結果、メッセージが表示されました。**hello.c**はごく簡単なプログラムですが、実行したときに**Hello World**と表示されれば、本書のほかの例についても問題なくコンパイルして実行することができるはずです。

最初の例ですから、いくつかの点について解説を加えておきましょう。まず、**hello**プログラムのある場所ですが、おそらくホームディレクトリでしょう。この場合、ホームディレクトリが**PATH**に含まれていないと、シェルは**hello**を見つけることができません。さらに、**PATH**に含まれるいずれかのディレクトリの中に、**hello**という名前でほかのプログラムが存在していた場合には、ホームディレクトリの**hello**ではなく、同名の別のプログラムが実行されることになります。このようなケースは、同名の**hello**プログラムを含むディレクトリが、**PATH**の中でホームディレクトリより前に指定されている場合にも発生します。

こうした問題を避けるには、プログラムを実行するときに、名前の前に**./**を付けて**./hello**のように指定します。こうすれば、現在のディレクトリにある指定された名前のプログラムを実行しなさい、とシェルに指示することになり、ホームディレクトリの**hello**を実行することができます。

1.3.3 ヘルプの表示

システムのプログラミングインタフェースと標準ユーティリティに関しては、どのUNIXシステムの場合でも比較的文書化が進んでいます。これは、最も初期のUNIXシステムのときから、プログラムに**マニュアルページ**を添付することが推奨されてきたためです。こうしたマニュアルページは、印刷された形で提供されている場合もありますが、ほとんどの場合、オンラインで参照することができます。

オンラインマニュアルページにアクセスするには、**man**コマンドを使います。マニュアルページの質や説明の詳しさは、項目によって大きく異なります。たとえば、より詳しい説明ののった

ほかの項目を参照させるようになっているものもあれば、プログラムのオプションとコマンドをすべて網羅し、これらについて詳細に解説したものもあります。いずれにしても、マニュアルページは第一に参照すべき情報源です。

GNUソフトウェアやほかのフリーソフトウェアでは、**info**と呼ばれるオンラインマニュアルシステムも使われています。このオンラインマニュアルシステムでは、**info**という特別なプログラム、または**emacs**エディタの**info**コマンドを使って、すべての情報をオンラインで参照することができます。**info**システムの優れているところは、リンクや相互参照を利用してマニュアルの中を自由に移動したり、目的の場所に直接ジャンプしたりできる点です。また、マニュアルの作者にとっても、同じ1つのソースからオンラインマニュアルだけでなく、印刷用の版組されたドキュメントまで作成できるという利点があります。

例題 マニュアルページとinfo

1 GNU Cコンパイラのマニュアルを表示してみましょう。まず、マニュアルページです。

```
$ man gcc

GCC(1)                    GNU Tools                    GCC(1)

NAME
       gcc, g++ - GNU project C and C++ Compiler (v2.7)

SYNOPSIS
       gcc [ option | filename ]...
       g++ [ option | filename ]...

WARNING
       The  information  in  this man page is an extract from the
       full documentation of the GNU C compiler, and  is  limited
       to the meaning of the options.

       This  man  page  is not kept up to date except when volun-
       teers want to maintain it.   If  you  find  a  discrepancy
       between  the  man  page and the software, please check the
       Info file, which is the authoritative documentation.

       If we find that the things in this man page that  are  out
       of date cause significant confusion or complaints, we will
       stop distributing the man page.  The alternative, updating
       the  man  page when we update the Info file, is impossible
       because the rest of the work of maintaining GNU CC  leaves
       us no time for that.  The GNU project regards man pages as
       obsolete and should not let them take time away from other
       things.

       For  complete and current documentation, refer to the Info
       file `gcc' or the manual Using and  Porting  GNU  CC  (for
       version  2.0).  Both are made from the Texinfo source file
```

```
        gcc.texinfo.

...
```

このマニュアルページでは、コンパイラがサポートしているオプションや、ターゲットプロセッサごとのオプションについての説明を読むことができます。マニュアルページとしてはかなり長い部類に入りますが、実際にはGNU C（およびC++）のマニュアル全体のほんの一部分を解説しているだけです。

2 GNU Cについてより詳しく知りたい場合には、**info**を使います。

```
$ info gcc

File: gcc.info,  Node: Top,  Next: Copying,  Up: (DIR)

Introduction
************

   This manual documents how to run, install and port the GNU compiler,
as well as its new features and incompatibilities, and how to report
bugs.  It corresponds to GNU CC version 2.7.2.

* Menu:

* Copying::             GNU General Public License says
                         how you can copy and share GNU CC.
* Contributors::        People who have contributed to GNU CC.
* Funding::             How to help assure funding for free software.
* Look and Feel::       Protect your freedom--fight "look and feel".

* G++ and GCC::         You can compile C or C++ programs.
* Invoking GCC::        Command options supported by `gcc'.
* Installation::        How to configure, compile and install GNU CC.
* C Extensions::        GNU extensions to the C language family.
* C++ Extensions::      GNU extensions to the C++ language.
* Trouble::             If you have trouble installing GNU CC.
* Bugs::                How, why and where to report bugs.
* Service::             How to find suppliers of support for GNU CC.
* VMS::                 Using GNU CC on VMS.

* Portability::         Goals of GNU CC's portability features.
* Interface::           Function-call interface of GNU CC output.
* Passes::              Order of passes, what they do, and what each file is for.
* RTL::                 The intermediate representation that most passes work on.
* Machine Desc::        How to write machine description instruction patterns.
* Target Macros::       How to write the machine description C macros.
* Config::              Writing the `xm-MACHINE.h' file.
* Fragments::           Writing the `t-TARGET' and `x-HOST' files.

* Index::               Index of concepts and symbol names.
```

```
--zz-Info: (gcc.info.gz)Top, 38 lines --All-- Subfile: gcc.info-1.gz-----------
Welcome to Info version 2.16. "C-h" for help, "m" for menu item.
```

このように、たくさんの項目がメニュー形式で表示されます。**info**システムでは、メニュー項目と階層的なページ構成によって、項目を選択するだけで広大なマニュアル全体の中を自由に移動できるようになっています。

infoシステムには、**info**システム自体のヘルプも（もちろん**info**形式で）用意されています。**Ctrl - H**を入力すれば、**info**の使い方のヘルプが表示されます。**info**プログラムは、多くのLinuxディストリビューションに付属しており、ほかのUNIXシステムにインストールすることもできます。

1.4 開発のためのロードマップ

UNIXで開発を行う場合、各種のツールや開発リソースがどこにあるかを知っておくことも重要です。以下では、いくつかの重要なディレクトリとファイルについて説明します。説明の具体的内容はLinuxを対象としていますが、その背後にある原則や考え方はほかのUNIXライクなシステムにもあてはまります。

1.4.1 プログラム

通常、プログラムは、そのために特別に用意されたディレクトリに置かれます。プログラム開発に使うものも含めて、システムが提供する汎用のプログラム群は**/usr/bin**に置かれます。特定のホストコンピュータまたはローカルネットワークを対象としてシステム管理者が追加したプログラムは、**/usr/local/bin**にあります。

管理者は、**/usr/local**というディレクトリをよく使います。これは、ベンダー提供のファイルやあとから追加したファイルをこのディレクトリの下に置くことによって、これらのファイルとシステムが提供するファイルとを明確に分離することができるからです。このような方針に従って**/usr**以下を管理しておけば、オペレーティングシステムをアップグレードする際も**/usr/local**を保存するだけでよく、余計な手間がかかりません。自分でコンパイルしたプログラムも、**/usr/local**以下で実行し、必要なファイルへのアクセスもこのディレクトリの下で行えるようにしておくとよいでしょう。

追加機能やプログラミングシステムの中には、独自のディレクトリ構造とプログラムディレクトリを持つものがあります。たとえば、X Window Systemは、一般に**/usr/X11**という名前のディレクトリにインストールされます。また、Revision 6の場合には**/usr/X11R6**、XFree86プロ

ジェクトによって配布され、多くのLinuxディストリビューションが採用しているIntelプロセッサ用XFree86の場合には/usr/X386というディレクトリも使われます。Solarisに付属するSun OpenWindowsシステムでは、/usr/openwinが使われます。

本書の最初のプログラムをコンパイルするときに使ったGNUコンパイラシステムのドライバプログラムgccは、通常、/usr/binまたは/usr/local/binにあります。ただし、gccはさまざまなコンパイラサポートプログラムを別の場所から実行します。この場所は、コンパイラ自体をコンパイルしたときに決定され、ホストコンピュータの種類によって異なります。Linuxシステムの場合には、/usr/lib/gcc-lib/i486-unknown-linux/2.7.2などに置かれるのが一般的です。GNU C/C++コンパイラとGNU固有のヘッダーファイルはこの場所にあります。

1.4.2 ヘッダーファイル

Cやその他の言語でプログラミングする場合、定数の定義や、システムとライブラリの関数呼び出しの宣言などが収められたヘッダーファイルが必要になります。Cの場合、これらのヘッダーファイルはほとんど常に/usr/includeディレクトリと、その下のサブディレクトリに置かれます。特定のUNIXまたはLinuxに依存するヘッダーファイルは、一般に/usr/include/sysと/usr/include/linuxに置かれます。

その他のプログラミングシステムにもヘッダーファイルがあり、これらのヘッダーファイルが置かれているディレクトリは、コンパイラの実行時に自動的に検索されます。たとえば、X Window Systemでは/usr/include/X11、GNU C++では/usr/include/g++などです。

Cコンパイラに対して-Iフラグを指定すれば、サブディレクトリや標準以外の場所にあるインクルードファイルを使用することができます。たとえば、次のようにします。

```
$ gcc -I/usr/openwin/include fred.c
```

この例では、fred.cプログラムでインクルードされているヘッダーファイルをコンパイラが探すときに、標準の場所に加えて/usr/openwin/includeディレクトリも検索されます。詳細については、Cコンパイラのマニュアルページを参照してください。

ヘッダーファイルから特定の定義や関数プロトタイプを検索するときには、grepコマンドを使うと便利です。たとえば、プログラムからの終了ステータスを返すのに使われる定数の名前を知りたいとします。この場合には、/usr/includeディレクトリに移動し、名前の一部に使われていそうな文字列を指定してgrepを実行します。

```
$ grep EXIT_ *.h
stdlib.h:#define     EXIT_FAILURE    1    /* Failing exit status.  */
stdlib.h:#define     EXIT_SUCCESS    0    /* Successful exit status.  */
$
```

grepは、ディレクトリ内の.hで終わる名前を持つすべてのファイルの中から、文字列EXIT_を検索します。上の出力からわかるように、目的の定義はstdlib.hにありました。

1.4.3 ライブラリファイル

ライブラリとは、再利用を目的として作成されたコンパイル済みの関数の集合のことです。通常、ライブラリは、よく行う作業を実行するための関連する関数から構成されています。たとえば、入出力関数のライブラリ（stdioライブラリ）や、データベースアクセスルーチン（dbmライブラリ）などがあります。これらのライブラリについては、あとの章でまた取り上げます。

通常、標準システムライブラリは/libと/usr/libに置かれています。Cコンパイラ（正確にはリンカ）は、デフォルトでは標準Cライブラリしか検索しないので、検索の対象に含めたいライブラリがある場合には明示的に指定する必要があります。これは、コンピュータが低速で、CPUサイクルが高価だった時代のなごりです。ライブラリを標準ディレクトリに置くだけでは、該当するライブラリが自動的に検索されることはありません。ライブラリは非常に特殊な名前付け規則に従っているので、コマンドラインでライブラリを指定する必要があります。

ライブラリの名前は、常にlibで始まります。libのあとには、なんのライブラリかを示す文字列が付きます。たとえば、Cライブラリの場合にはc、数学ライブラリの場合にはmです。ドット（.）で始まるライブラリ名の最後の部分は、次に示すようにライブラリの種類を表します。

- .aは伝統的な**静的ライブラリ**（後述）です。静的ライブラリはスタティックライブラリと訳されることもあります。
- .soと.saは**共有ライブラリ**（後述）です。共有ライブラリはシェアードライブラリと訳されることもあります。

ls /usr/libを実行するとわかるように、通常、ライブラリには静的ライブラリと共有ライブラリの両方の形式が存在します。検索するライブラリをコンパイラに指定するときには、ライブラリのフルパス名を指定するか、または-lフラグを使います。次の例ではフルパス名を指定しています。

```
$ cc -o fred fred.c /usr/lib/libm.a
```

この例では、ファイルfred.cがコンパイルされ、その結果得られるプログラムファイルの名前がfredになり、関数への参照を解決するために標準Cライブラリに加えて数学ライブラリが検索されます。

次のように指定しても、まったく同じ結果が得られます。

```
$ cc -o fred fred.c -lm
```

-lm（**l**と**m**の間には空白を入れないこと）は、いずれかの標準ライブラリディレクトリ（この場合には**/usr/lib**）にある**libm.a**という名前のライブラリを短縮形で指定しています。UNIXでは短縮形が好まれる傾向がありますが、**-lm**という形式を使うと、共有ライブラリが存在する場合には自動的に共有ライブラリが使われるという利点もあります。

通常、ライブラリはヘッダーファイル同様、標準的な場所に置かれていますが、**-L**フラグ（大文字であることに注意）を使って検索ディレクトリを追加することもできます。たとえば、次のように指定します。

```
$ cc -o x11fred -L/usr/openwin/lib x11fred.c -lX11
```

この例では、ディレクトリ**/usr/openwin/lib**にあるライブラリ**libX11**を使って、**x11fred**という名前のプログラムのコンパイルとリンクを行います。

-o nameオプションが省略された場合、コンパイラは、作成したプログラムを**a.out**という名前のファイルに出力します。**a.out**とは、アセンブラ出力（assembler output）の意味です。プログラムをコンパイルしたはずなのに、できあがったプログラムが見つからない場合には、**a.out**というファイルがないかどうか調べてください。ちなみに、UNIXが誕生して間もないころ、システムでゲームをして遊びたかった人たちは、システム管理者に見つからないように**a.out**という名前でゲームを実行していました。また、大規模なUNIXシステムでは、毎朝**a.out**という名前のファイルをすべて削除するようにしていることがあります。

◆ 静的ライブラリ

最も単純な形式のライブラリは、すぐに使える状態でまとめられたオブジェクトファイルの集合です。ライブラリに格納されている関数をプログラムで使う場合には、その関数を宣言しているヘッダーファイルをインクルードします。プログラムコードとライブラリがコンパイラとリンカによって結合され、1つの実行可能プログラムが生成されます。

静的ライブラリは**アーカイブ**とも呼ばれ、慣習上、**.a**で終わる名前を持っています。たとえば、**/usr/lib/libc.a**と**/usr/X11/lib/libX11.a**は、それぞれ標準Cライブラリと X11ライブラリです。

アーカイブ（archive）を縮めて**ar**と命名されたプログラムを使用し、**cc -c**で関数を個別にコンパイルすれば、独自の静的ライブラリを簡単に作成、保守することができます。複数の関数から共通のデータにアクセスする必要がある場合には、これらの関数を同じソースファイルにまとめ、そのソースファイルでstatic変数を宣言します。

例題 静的ライブラリ

1 2つの関数を含む小さな独自のライブラリを作成し、ライブラリの関数を使ってプログラムをコンパイルしてみましょう。ライブラリに収める関数の名前は`fred`と`bill`で、それぞれ簡単なメッセージを表示します。まず、2つの関数に対応するソースファイルを個別に作成します。名前は関数に合わせてそれぞれ`fred.c`、`bill.c`とします。

```
#include <stdio.h>

void fred(int arg)
{
    printf("fred: you passed %d\n", arg);
}
```

```
#include <stdio.h>

void bill (char *arg)
{
    printf("bill: you passed %s\n", arg);
}
```

上の2つの関数を個別にコンパイルすれば、ライブラリに追加できる状態のオブジェクトファイルを作成することができます。それには、最終的なプログラムの作成を抑止する`-c`オプションを指定してCコンパイラを呼び出します（上の例では、最終的なプログラムを作成しようとしても、`main`という名前の関数が定義されていないのでエラーが発生します）。

```
$ cc -c bill.c fred.c
$ ls *.o
bill.o   fred.o
```

2 次に、関数`bill`を呼び出すプログラムを書いてみましょう。実際にプログラムを記述する前に、ライブラリ用のヘッダーファイルを用意しておくと便利です。このヘッダーファイルでは、ライブラリに含まれる関数の宣言を記述します。ライブラリを使うプログラムでは、このヘッダーファイルをインクルードします。

```
/*
    This is lib.h. It declares the functions fred and bill for users
*/

void bill(char *);
void fred(int);
```

3 関数`bill`を呼び出すプログラム（ここでは`program.c`という名前にします）は、非常にシンプルに記述することができます。次に示すように、ライブラリヘッダーファイルをインクルードし、ライブラリの中の関数を1つ呼び出すだけです。

```
#include "lib.h"

int main()
{
    bill("Hello World");
    exit(0);
}
```

さっそくプログラムをコンパイルしてテストしましょう。まず、オブジェクトファイルを明示的に指定する方法を使って、上のプログラムをコンパイルしたあと、前の手順でコンパイルしておいたオブジェクトモジュール**bill.o**とリンクします。

```
$ cc -c program.c
$ cc -o program program.o bill.o
$ program
bill: you passed Hello World
$
```

4 今度はライブラリを作成し、実際に使ってみましょう。アーカイブを作成し、作成したアーカイブにオブジェクトファイルを追加するには、**ar**というプログラムを使います。**ar**という名前は、アーカイブ、つまり1つのファイルに収められた複数のファイルの集合を作成するのに使われることから付けられたものです。**ar**は、ほかの多くのUNIXユーティリティと同様に非常に汎用的なツールなので、**ar**を使ってテキストファイルのアーカイブを作成することもできます。

```
$ ar crv lib.a bill.o fred.o
a - bill.o
a - fred.o
```

ライブラリが作成され、2つのオブジェクトファイルがライブラリに追加されます。ライブラリを使う際、システムによっては(特にBerkeley UNIXから派生したシステムの場合)、ライブラリの内容に関するテーブルの作成が必要になることがあります。テーブルを作成するには、**ranlib**コマンドを使います。Linuxのように GNU ソフトウェア開発ツールを使っている場合には、この手順は不要です(実行しても害はありませんが)。

```
$ ranlib lib.a
```

これでライブラリが使用できるようになりました。ライブラリを使ってプログラムを作成するには、次のようにします。

```
$ cc -o program program.o lib.a
$ program
bill: you passed Hello World
$
```

オブジェクトファイルやライブラリ、実行可能プログラムにどのような関数が含まれているかを調べるには、nmコマンドを使います。たとえば、programとlib.aを調べてみると、ライブラリにはfredとbillの両方が含まれているのに対して、programにはbillしか含まれていないことがわかります。プログラムが作成されるときには、そのプログラムで実際に必要になる関数だけがライブラリから引き出されて使われます。ライブラリに含まれるすべての関数の宣言を記述したヘッダーファイルをインクルードしても、作成されるプログラムにライブラリの全関数が含まれるわけではありません。

MS-DOSやMicrosoft Windowsでソフトウェア開発をした経験がある場合には、次の対応表を参考にしてください。

表1.1　UNIXとDOS/Windowsとの対応

要素	UNIX	DOS/Windows
オブジェクトモジュール	func.o	FUNC.OBJ
静的ライブラリ	lib.a	LIB.LIB
プログラム	program	PROGRAM.EXE

◆ 共有ライブラリ

静的ライブラリのひとつの欠点は、同じライブラリの関数を使ったプログラムを同時に多数実行した場合に、同じ関数のたくさんのコピーがメモリに置かれることです。これらのコピーは、プログラムファイル自体にも含まれています。このため、静的ライブラリを使うと、場合によっては貴重なメモリとディスクスペースを無駄に消費することになります。

多くのUNIXシステムは、こうした静的ライブラリの欠点を克服するために、共有ライブラリをサポートしています。共有ライブラリの詳細や各種システム上での実装についての説明は本書の扱う範囲を超えるので、ここではLinuxでの実装について簡単に説明します。

共有ライブラリは、静的ライブラリと同じ場所に置かれていますが、別の拡張子が付けられています。一般的なLinuxシステムでは、共有バージョンの標準Cライブラリは/lib/libc.so.Nです。ここで、Nはメジャーバージョン番号（本書の執筆時点では5）を表します。

共有ライブラリを使うプログラムは、スタブライブラリとリンクされます。たとえば、標準Cライブラリの場合、スタブライブラリは/usr/lib/libc.saになります。スタブライブラリは特殊な種類のライブラリで、関数のコード自体は含まれておらず、実行時に利用可能になる共有コードへの参照が含まれています。作成したプログラムがメモリに読み込まれて実行されると、関数への参照が解決され、共有ライブラリに対して呼び出しが行われます。このとき、必要に応じて共有ライブラリがメモリに読み込まれます。

このしくみによって、共有ライブラリの1つのコピーを多くのアプリケーションで同時に使用でき、また、ディスク上にも1つのコピーを保存しておくだけですむようになっています。共有ライブラリは、ライブラリを使うプログラムとは別個に更新できるという利点もあります。マイナーバージョン番号のリビジョンは、クライアントプログラムにとって意味を持ちません。ファ

イル/lib/libc.so.5から実際のマイナーバージョン（執筆時点では/lib/libc.so.5.4.38）へのシンボリックリンクが使われるからです。

Linuxシステムの場合、共有ライブラリのロード、およびクライアントプログラムの関数参照の解決を行うプログラムは、ld.soまたはld-linux.soです。共有ライブラリを検索する場所は、ファイル/etc/ld.so.confで設定されています。X11の共有ライブラリを追加した場合など、/etc/ld.so.confファイルを変更したときには、ldconfigを使って変更内容を有効にする必要があります。

lddというユーティリティを使うと、プログラムが使う共有ライブラリを調べることができます。

```
$ ldd program
        libc.so.5 => /lib/libc.so.5 (0x4000a000)
```

上の例では、標準Cライブラリ（libc）は共有ライブラリ（libc.so）です。プログラムが必要とするメジャーバージョンは5で、/lib/libc.so.5が使われます。ほかのUNIXシステムでも、同じような方法で共有ライブラリにアクセスしています。興味があれば、システムのマニュアルを参照してみるとよいでしょう。

共有ライブラリは、多くの点でMicrosoft Windowsで使われるダイナミックリンクライブラリと似ています。この場合、.DLLファイルに対応するのは.soライブラリで、.saライブラリは.LIBファイルに相当します。

1.5 UNIXの哲学

本書の以下の章は、UNIXプログラミングの特徴や流儀を読者につかんでもらうことを意図して書かれています。たしかにCのプログラミング自体は、プラットフォームが異なっていても同じです。しかし、UNIXの開発者たちは、プログラムやシステム開発に関して独特の考え方を持っています。

UNIXオペレーティングシステムには、重要とされるプログラミングパラダイム、あえていえば一種の哲学のようなものがあります。一般にUNIXのプログラムとシステムでは、次のようなことが大切だと考えられています。

◆ シンプル

便利なUNIXユーティリティのほとんどは非常にシンプルで小さく、習得も容易です。こうした設計思想を支えているのが、“Keep It Simple, Stupid”、略して**KISS**という考え方です。これは、実際のプログラミングの際にもお手本とすべき優れたテクニックです。システムが大きく複

雑になればなるほど、混入するバグの数やその複雑さも増し、デバッグ作業にもたいへんな労力が必要になるからです。

◆ 目的の明確化

多くの場合、1つのプログラムでは1つの作業だけを実行するほうがソリューションとして優れています。あまりに多くの機能を詰め込んだプログラムは使いにくく、保守にも手間がかかります。目的を1つに絞り込んで作成したプログラムは、あとでよりよいアルゴリズムやインタフェースが開発されたときに容易に改善することができます。UNIXでは、ユーザーのニーズをあらかじめ取り込んだ形で1つの大きなプログラムを作るのではなく、必要に応じて小さなユーティリティをいくつも組み合わせて複雑な作業を実行するほうが一般的です。

◆ 再利用可能な要素

アプリケーションのコアとなる部分をライブラリとして利用できるようにします。シンプルで柔軟なプログラミングインタフェースを持ち、きちんと文書化されたライブラリを用意しておくと、ほかのユーザーが同種のプログラムを開発したり、ライブラリに組み込まれたテクニックを新しいアプリケーションに応用したりするときに、大いに役立ちます。単なるデータベース管理プログラムとしてではなく、再利用可能な関数の集合として提供されている`dbm`データベースライブラリなどは、そのよい例です。

◆ フィルタ

UNIXには、**フィルタ**として使用できるアプリケーションが数多くあります。フィルタとは、入力内容を変換して出力を生成するプログラムのことです。UNIXでは、既存のいくつかのプログラムを斬新な方法で組み合わせることによって複雑なアプリケーションを作成することができます。こうしたプログラムの再利用が可能なのも、上に述べたような開発姿勢があるためです。

◆ オープンなファイル形式

よく使われる人気の高いUNIXプログラムは、設定ファイルとデータファイルにプレーンなASCIIテキストファイルを使用しています。プログラムを開発するときには、このファイル形式の採用を心がけてください。ASCIIテキストファイルを使えば、ユーザーも標準的なツールを使って設定項目を変更したり検索したりできます。また、データファイルがASCIIテキストファイルになっていれば、これらのファイルに対してなにか新しい処理を行う場合でも、そのためのツールを容易に開発することができます。こうした原則に基づいて作成されたプログラムの例として、シンボルの位置に関する情報を各種検索プログラムに適した正規表現の形で記録する`ctags`ソースコードクロスリファレンスシステムなどがあります。

◆ 柔軟性

ユーザーがプログラムをどのように使うかは、プログラムを作成した本人にも予想できません。したがって、プログラムを作成するときには、できるだけプログラムに柔軟性を持たせることが大事です。たとえば、フィールドサイズやレコード数などに対して、「これくらいあればいいだろう」というような根拠のない制限は設けないようにします。また、可能であれば、ネットワークを意識し、ローカルマシン上だけでなくネットワーク経由でも動作するようにプログラムを作成します。ユーザーがなにをするかは、プログラムを作成する時点ですべて予測可能であるというような過信は禁物です。

1.6 この章のまとめ

この章では、Linuxと商用UNIXシステムとの共通点について説明し、UNIXの開発者が利用できるプログラミングシステムにさまざまなものがあることを示しました。

また、簡単なプログラムとライブラリを作成してCの基本ツールの使い方を示し、MS-DOS上のツールとの関係についても簡単に触れました。最後に、UNIXのプログラミングパラダイムについて解説しました。

第2章

シェルプログラミング

CによるUNIXプログラミングを取り上げたばかりですが、この章では少しまわり道をしてシェルプログラミングについて学びます。

シェルの使い方を覚えると、環境を豊かにすることができます。シェルはDOSのコマンドプロセッサ`command.com`と似ていますが、はるかに強力であり、立派なプログラミング言語といえます。コマンドを実行したり、UNIXユーティリティを呼び出したりできるだけでなく、コマンドやユーティリティ自体をシェルで記述することもできます。また、インタープリタ型の言語なので、行単位で実行でき、デバッグも容易です。もちろん、再コンパイルなどは必要ありません。ただし、こうしたシェルの特性ゆえの欠点もあります。それは、タイムクリティカルなタスクやプロセッサを酷使するようなタスクには向かないことです。

シェルを使ったプログラミングにはどのような利点があるのでしょうか。まず、シェルを使えば、手軽にプログラムを作成することができます。また、シェルは、ミニマムインストールのUNIXでも常に利用可能です。簡単なプロトタイプ作成に使えば、自分のアイディアをプログラムとして実装できるかどうか見当をつけることもできます。シェルは、比較的単純なタスクを実行する小さなユーティリティを作成するのに適しています。特に、効率や速度よりも、設定や保守の容易性、高い移植性が求められる場合には最適です。シェルではプロセス制御を扱うこともできるので、複数のコマンドを実行する場合でも、各ステップが正常に終了したかどうかに応じて、あらかじめ指定した順序でコマンドを実行することができます。

メモ

使用しているシステムにも、多くのシェルスクリプトの例があるはずです。たとえば、`.xinitrc`や`startx`、起動時のシステム構成を行う`/etc/rc.d`ディレクトリのスクリプト群、Free Software Foundationのパッケージインストーラ`autoconf`などはシェルスクリプトです。興味があったら覗いてみるとよいでしょう。

前の章の最後でUNIXの哲学について解説しました。シェルプログラミングを学ぶと、実際にUNIXの哲学の一端に触れることができます。UNIXは高いレベルでのコードの再利用を前提とし、またそれに依存しています。たとえば、だれかが小さいシンプルなユーティリティを作ると、それをほかのユーザーが別のユーティリティと組み合わせて新しいコマンドとして使います。ごく単純な例を次に示します。

```
$ ls -al | more
```

この例では、`ls`と`more`というユーティリティを組み合わせて、ファイル一覧の出力をパイプに渡して結果を1画面ごとに表示します。UNIXではどんなユーティリティでも、上のようなコマンドの要素として利用することができます。いってみれば、UNIXプログラマはおもちゃのレゴ遊びがとても好きなのです。レゴのひとつひとつは小さな部品にすぎませんが、これを巧みに組み合わせることでさまざまなものを作れるように、UNIXでも小さなスクリプトをたくさん寄せ集めることで、大きく複雑なプログラムを作ることができます。

メモ

たとえば、**bash**の**man**ページを印刷したい場合には、**man bash | col -b | lpr**を実行します。

さらに、前の章で説明したように、UNIXではファイルの属性によって実行可能かどうかが判断されるので、ユーティリティを使うユーザーは、そのユーティリティがどの言語で書かれているかを気にする必要がありません。ユーティリティの処理速度を改善したい場合には、シェルでプロトタイプを作成し、プログラムとして機能するかどうかを確かめたうえで、あとからCやC++で実装し直すことができます。逆に、シェルのままでも十分目的を達成できる場合には、そのままにしておけばよいのです。

メモ

CやC++の代わりとしてよく使われるその他のインタープリタ系言語には、Perl、Tcl/Tk、Pythonなどがあります。これらの言語の場合にも、インタープリタによって処理されるスクリプトか、コンパイル済みプログラムかの違いはそれほど意味を持ちません。事実、Tclでは最近になってコンパイラを含むバージョンがリリースされています。

作成したスクリプトをCやC++で実装し直すべきかどうかは、最適化や移植の必要性、設定変更の柔軟性、目的とのバランスなどを考慮して判断する必要があります。

システム管理でシェルスクリプトを扱わなければならない場合はもちろん、新しく浮かんだアイディアをとりあえず試してみたい場合や、何度も繰り返し実行する作業を手際よく処理したい場合などにも、シェルプログラミングの知識が大いに役立ちます。

この章では、シェルプログラミングの構文、制御構造、コマンドについて学びます。取り上げる例は網羅的といえるほど多くはありませんが、シェルの機能と威力を十分に実感できるはずです。また、この章の最後では、実際の使用にも十分たえるスクリプトを作成します。このスクリプトは、あとの章でCを使って書き直し、機能を追加します。

この章で取り上げる項目は次のとおりです。

- シェルとは
- シェルの基本
- シェルの構文、変数、条件、プログラムの制御
- リスト
- 関数
- シェルのコマンドとコマンド実行の構文
- ヒヤドキュメント
- デバッグ

2.1 シェルとは

まずシェルの機能と、UNIXで利用できるさまざまなシェルの種類について見ておきましょう。

シェルとは、ユーザーとUNIXシステムの間のインタフェースとして機能するプログラムのことです。シェルが存在することで、ユーザーはコマンドを入力し、これをオペレーティングシステムに実行させることができます。

そうした意味では、シェルはDOSに似ています。しかし、シェルは、カーネルで行われる処理の細かな部分をユーザーから隠す役割も果たしています。たとえば、ファイルのリダイレクトで使う`<`と`>`、パイプを表す`|`、サブプロセスからの出力で置き換えられる`$(...)`などは、どのように実装されているかを知らなくても使うことができます。このような点に注目すれば、シェルはUNIX用の高級プログラミング言語ということができます。

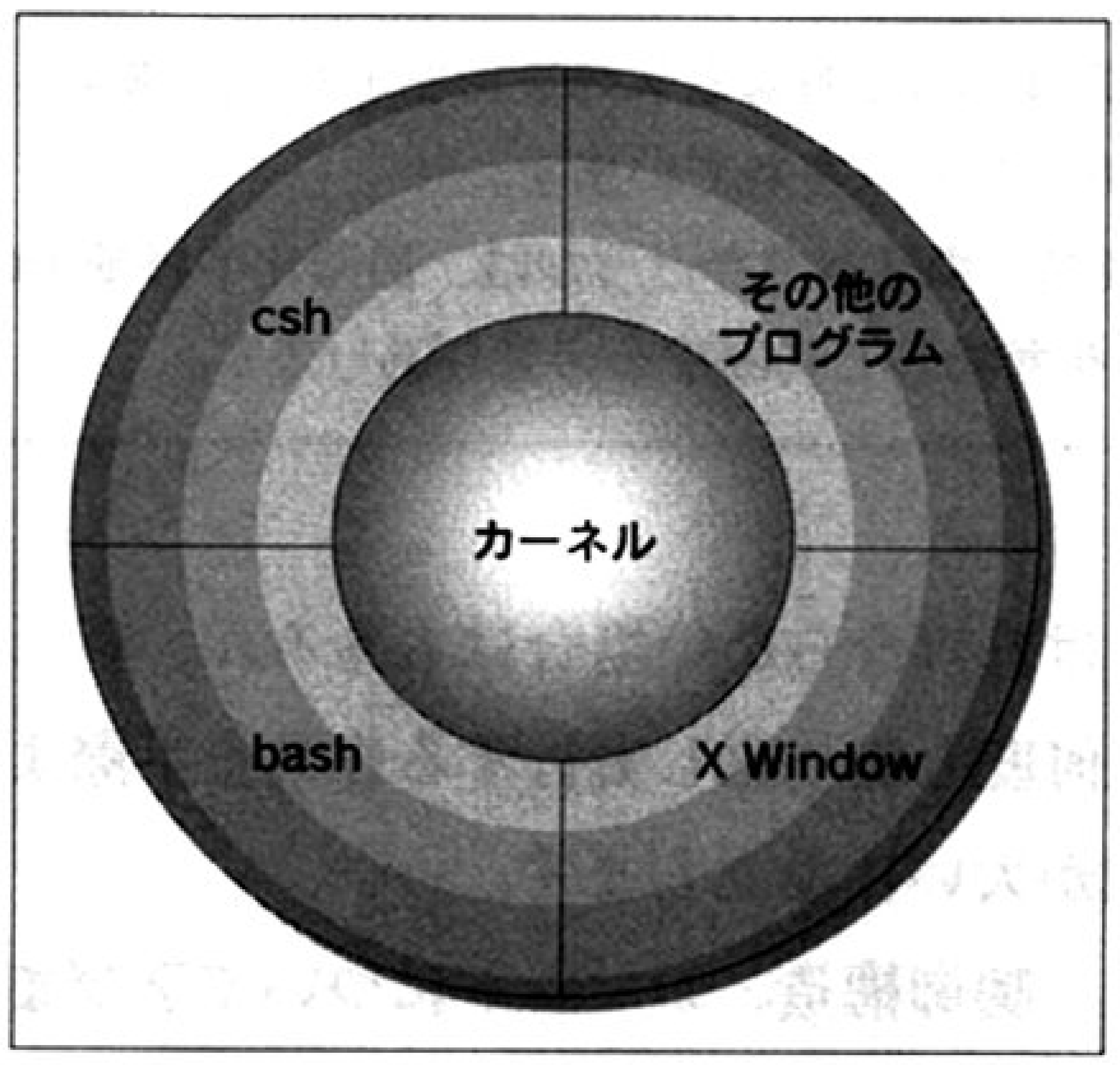


図2.1 シェルの位置づけ

UNIXではモジュール化がかなり進んでおり、シェルもさまざまな種類の中から好みのものを選ぶことができます。現在利用できるシェルのほとんどは、オリジナルのBourneシェルから派生したものです。

表2.1 シェルの種類

シェル	説明
`sh` (Bourne)	オリジナルシェル。
`csh`、`tcsh`、`zsh`	Berkeley UNIXで有名なBill Joyが作成したCシェルと、その流れを汲むシェル。`bash`に次いで人気がある。
`ksh`、`pdksh`	Kornシェルとそのパブリックドメイン版。David Kornが作成。
`bash`	GNUプロジェクトから生まれたシェルで、Linuxの標準シェル。`bash`という名前は、Bourne Again Shellの略。ソースコードが入手可能で、数多くのUNIXシステムに移植されている。
`rc`	`csh`よりもさらにCに近いシェル。これもGNUプロジェクトの成果。

Cシェルを除けば、どのシェルも互いに非常によく似ており、X/Open 4.2およびPOSIX 1003.2仕様で指定されたシェルに近いものになっています。POSIX 1003.2ではシェルのミニマムな仕様を定めていますが、X/Openではこれを拡張し、より使いやすい強力なシェルを実現可能にしています。一般にX/Openのほうが仕様の要求水準は高いのですが、それだけユーザーフレンドリなシステムの構築が可能になっています。

この章では、主にPOSIX互換シェルに共通する機能を取り上げます。また、シェルは、デフォルトで**/bin/sh**としてインストールされているものと仮定します。

メモ

GNUプロジェクトでは、一連の基本コマンドを収めたシェルユーティリティ**sh-utils**を配布しており、このパッケージを利用すると、システムに付属のものより高いパフォーマンスが得られることがあります。また、シェルスクリプトをアーカイブしたい場合には、**shar**パッケージを利用すると便利です。

2.2 パイプとリダイレクト

シェルプログラムの説明に入る前に、UNIXプログラム（シェルプログラム以外のプログラムも含む）の入力と出力のリダイレクトについて簡単に触れておきましょう。

2.2.1 出力のリダイレクト

リダイレクトのうち、いくつかのものはすでに使ったことがあるはずです。たとえば、次の例を見てください。

```
$ ls -l > lsoutput.txt
```

この例では、**ls**コマンドからの出力が**lsoutput.txt**という名前のファイルに保存されます。

これはごく単純な例であり、実際にはもっと複雑なリダイレクトが可能です。標準ファイルデスクリプタについては第3章で詳しく学びますが、ここではとりあえず、ファイルデスクリプタ0がプログラムへの**標準入力**、ファイルデスクリプタ1が**標準出力**、ファイルデスクリプタ2が**標準エラー出力**を表すことを覚えておいてください。これらのファイルデスクリプタは、それぞれ個別にリダイレクトできます。実際には、ほかのファイルデスクリプタもリダイレクトできますが、標準の0、1、2以外のファイルデスクリプタをリダイレクトする必要がある場合はまれでしょう。

上に示した例では、**>**演算子を使って標準出力をファイルにリダイレクトしました。この場合、デフォルトでは、ファイルが存在すると、そのファイルは上書きされます。デフォルトの動作を

変えたい場合には、set -Cコマンドを使えば、リダイレクトでのファイルの上書きを禁止するnoclobberオプションを設定することができます。setコマンドについては、あとでさらに詳しく説明します。

出力をファイルに追加するには、>>演算子を使います。

```
$ ps >> lsoutput.txt
```

この例では、psコマンドの出力がファイルに追加されます。

標準エラー出力をリダイレクトするには、リダイレクトするファイルデスクリプタの値を>演算子の前に付けます。標準エラー出力のファイルデスクリプタは2ですから、ここでは2>のように指定することになります。標準エラー出力のリダイレクトは、エラー情報を画面に表示させないようにするときに便利です。

たとえば、killコマンドを使ってスクリプトからあるプロセスをkillするとしましょう。この場合、killコマンドを実行する前に目的のプロセスが死んでいることもあります。この状態でkillコマンドを実行すると、標準エラー出力にエラーメッセージが書き込まれ、通常はこれが画面に表示されます。標準出力と標準エラー出力の両方をリダイレクトすれば、killコマンドからのメッセージを画面に表示しないようにすることができます。たとえば、次のようにします。

```
$ kill -1 1234 > killout.txt 2>killerr.txt
```

この例では、コマンドの出力とエラー情報がそれぞれ別のファイルに書き込まれます。

両方の出力を1つのファイルに収めたい場合には、次のように>&演算子を使います。

```
$ kill -1 1234 > killouterr.txt 2>&1
```

これで、コマンドからの出力とエラー出力の両方が同じファイルに書き込まれます。ただし、このときに演算子の順序に注意する必要があります。上の例は、「標準出力をファイルkillouterr.txtにリダイレクトし、次に標準エラー出力を標準出力と同じ場所にリダイレクトする」という意味です。演算子の順序を間違えると、期待どおりの結果が得られないことがあるので注意が必要です。

killコマンドが成功したかどうかの結果はリターンコードで知ることができるので（リターンコードについてはあとで説明します）、実際には標準出力や標準エラー出力を保存する必要はないでしょう。そこで、UNIXの万能“ガラクタバケツ”である/dev/nullを使って次のように記述すれば、出力をすべて捨てることができます。

```
$ kill -1 1234 > /dev/null 2>&1
```

2.2.2 入力のリダイレクト

出力だけでなく、入力もリダイレクトすることができます。たとえば、次の例を見てください。

```
$ more < killout.txt
```

もっとも、これはUNIXではあまり意味のない例かもしれません。DOSのmoreコマンドと異なり、UNIXのmoreコマンドは引数にファイル名を受け付けるからです。

2.2.3 パイプ

パイプ（|）演算子を使うと、複数のプロセスを連結することができます。DOSとは異なり、UNIXではパイプで連結されたプログラムは同時に実行することが可能です。パイプで連結されたプロセスのスケジューリングは、プロセス間をデータが流れるに従って自動的に調整されます。

簡単な例として、psからの出力をsortコマンドでソートする場合を考えてみましょう。

パイプを使わない場合には、次のようにいくつかの手順に分けて処理を実行する必要があります。

```
$ ps > psout.txt
$ sort psout.txt > pssort.out
```

しかし、パイプを使ってプロセスを連結すれば、もっとエレガントに処理することができます。

```
$ ps | sort > pssort.out
```

おそらく1画面ごとに結果を表示したいでしょう。そこで、もうひとつ、moreのプロセスを加えてすべてを1行のコマンドラインで記述すると、次のようになります。

```
$ ps | sort | more
```

連結するプロセスの数には、実質的に制限はありません。たとえば、シェルを除くすべての実行中のプロセスのうち、プロセス名が異なるものだけを表示したい場合には、次のようにします。

```
$ ps -a | cut -c 21- | sort | uniq | grep -v sh | more
```

この例では、psの出力からcutでプロセス名の部分を切り出し、それをアルファベット順にソートして、uniqを使って重複行を取り除き、さらにgrep -v shを使ってshという名前のプロセスを除外し、その結果を1画面ごとに表示します。

以上で、シェルの基本操作についての説明は終わりです。さっそく本題のスクリプトの学習を始めましょう。

2.3 プログラミング言語としてのシェル

シェルプログラムを作成するには2つの方法があります。ひとつは、コマンドを入力し、シェルによってコマンドを対話的に実行させる方法です。もうひとつは、コマンドをファイルに保存し、このファイルをプログラムとして呼び出す方法です。

2.3.1 対話的プログラム

数行程度の短いコードをすばやく簡単に試すには、コマンドラインでシェルスクリプトを入力する方法が適しています。

> 注意
> システム標準のシェルが**bash**ではない場合など、現在使っているシェルを別のシェルに切り替えたいときには、シェル名（**/bin/bash**）をそのまま入力すれば、新しいシェルが実行され、コマンドプロンプトも変更されます。

ひとつの例として、Cのファイルがたくさんあり、その中から**POSIX**という文字列を含むファイルの内容を表示したいとします。**grep**コマンドを使ってファイルから目的の文字列を検索し、その結果得られたファイルを1つずつ表示する方法もありますが、必要な処理を次のような対話的スクリプトで実行することもできます。

```
$ for file in *
> do
> if grep -l POSIX $file
> then
> more $file
> fi
> done
posix
This is a file with POSIX in it - treat it well
$
```

シェルのコマンドを入力すると、通常の**$**というシェルプロンプトが**>**に変わることに注意してください。この場合、続けてコマンドを入力していけば、シェルが入力の終わりを自動的に判断してスクリプトを実行します。

上の例では、**POSIX**という文字列を含むファイルの名前が**grep**コマンドによって出力され、次に**more**によってファイルの内容が画面に表示されます。最後に、シェルプロンプトが戻ってきます。また、スクリプトの処理内容をわかりやすくするために、各ファイルを表すためのシェル変数は**file**としています。

シェルはワイルドカードの展開（グロビングともいいます）も行います。ワイルドカードには、上のスクリプトで使った*以外にも、任意の1文字を表す?、指定した文字集合のうちの任意の文字にマッチする[set]、指定した文字集合以外の任意の文字にマッチする[!set]などがあります。bashをはじめ、いくつかのシェルで使用できる中かっこ（{}）の展開を使うと、任意の文字列をグループ化し、これをシェルに展開させることができます。

```
$ ls my_{finger,toe}s
```

上の例では、ファイル名の一部が共通する2つのファイルが表示されます。最初に出した例では、シェルを使って現在のディレクトリ内のすべてのファイルをチェックしました。

熟練したUNIXユーザーなら、おそらく次のようなコマンドを入力し、同じ処理を効率的に実行することでしょう。

```
$ more `grep -l POSIX *`
```

次のコマンドも、構文的には同じです。

```
$ more $(grep -l POSIX *)
```

では、次のように入力した場合はどうでしょうか。

```
$ grep -l POSIX * | more
```

最後の例は、POSIXというパターンに一致する内容を持つファイルの名前を出力します。これらの例からわかるように、シェルではgrepやmoreなど、ほかのコマンドを使ってより複雑な作業を実行することができます。シェルを使うと、既存のコマンドをつなぎ合わせて新しい強力な機能を持たせることができます。

しかし、一連のコマンドを実行するときに、常にコマンドをひとつひとつ入力しなければならないとしたらとても面倒です。このような場合、コマンドをファイルに保存しておけば、必要に応じていつでもコマンドを実行することができます。コマンドを保存したファイルは、慣習上、**シェルスクリプト**と呼ばれています。

2.3.2 スクリプトの作成

まず、テキストエディタを使って、コマンドを記述したファイルを作成します。次のような内容のfirst.shというファイルを作成してください。

```
#!/bin/sh

# first.sh
# This file looks through all the files in the current
# directory for the string POSIX, and then prints those
# files to the standard output.

for file in *
do
  if grep -q POSIX $file
  then
    more $file
  fi
done

exit 0
```

#から行末まではコメントです。ただし、習慣的に#は行頭に置かれることが多いようです。#で始まる部分はコメントといいましたが、実は1行目の#!/bin/shは特別な形のコメントです。具体的には、#!のあとに続く引数によって、ファイルを実行するのに使うプログラムを指定します。たとえば、上の例では/bin/shがデフォルトのシェルプログラムになります。

このコメントでは絶対パスが指定されていることに注意してください。UNIXの実装によっては、インタープリタのパス長が32文字に制限されていることがあります。このため、よく使うシェルは/binにシンボリックリンクを張っておくほうが無難です。長い名前のコマンドや、深く掘られたディレクトリに置かれたコマンドを呼び出す指定になっていると、スクリプトが正しく動作しないことがあるからです。

スクリプトは基本的にシェルへの標準入力として扱われるので、スクリプトには、PATH環境変数に基づいて参照可能な任意のUNIXコマンドを記述することができます。

スクリプトの最後のexitコマンドは、スクリプトから意味のある終了コードを返します（終了コードについてはあとで詳しく説明します）。プログラムを対話的に実行する場合には、終了コードをチェックすることはほとんどありませんが、スクリプトを別のスクリプトから呼び出して正常に終了したかどうかを調べる場合には、適切な終了コードを返すようにしておく必要があります。さらにいうなら、たとえスクリプトをほかから呼び出すことを想定していなくても、きちんと終了コードを返すようにしておくべきです。これは、UNIX哲学の重要な要素である再利用の観点からも望ましいことです。自分の作成するスクリプトの有用性に自信を持って、きちんと終了コードを返す習慣を身につけておきましょう。

シェルプログラミングでは、0は正常に終了したことを意味します。上のスクリプトでは、エラーを検出することができないので、常に成功を意味する0を返します。成功した場合に終了コード0を使う理由については、あとでexitコマンドを取り上げるときに説明します。

2.3.3 スクリプトを実行可能にする

作成したスクリプトファイルを実行するには2つの方法があります。最も簡単なのは、次のようにスクリプトファイルの名前を引数としてシェルを呼び出す方法です。

```
$ /bin/sh first.sh
```

この方法でも実行は可能ですが、ほかのUNIXコマンド同様、ファイルの名前を入力するだけでスクリプトを実行できたほうが便利です。

それには、すべてのユーザーがファイルを実行できるように、**chmod**コマンドでファイルのモードを変更します。

```
$ chmod +x first.sh
```

メモ

chmodコマンドでファイルを実行可能にするには、上に示したやり方以外にもさまざまな方法があります。**man chmod**を実行し、8進数の引数やその他のオプションについても調べてみてください。

ファイルのモードを変更したあとは、次のようにしてスクリプトを実行することができます。

```
$ first.sh
```

注意

上のように入力しても、コマンドが見つからないという意味のエラーメッセージが表示されるかもしれません。これはほとんどの場合、シェルの環境変数**PATH**に現在のディレクトリが含まれていないことが原因です。**PATH=$PATH":."**とするか、または**.bash_profile**ファイルを編集して現在のディレクトリが**PATH**に含まれるようにしてください（もちろん、スクリプトが存在するディレクトリで、ファイルへの相対パスを指定して**./first.sh**と入力する方法もあります）。

ただし、**root**ユーザーの**PATH**変数には、現在のディレクトリを含めるべきではありません。不用意にこうした設定を行うと、**root**としてログインしたシステム管理者が、標準コマンドの代わりに、現在のディレクトリに置かれた偽のコマンドを呼び出してしまう危険性があり、セキュリティホールになります。

スクリプトが正しく動作することを確認したら、現在のディレクトリではなく、より適切な場所にスクリプトを移動することができます。作成したスクリプトが自分専用のコマンドである場合には、ホームディレクトリに**bin**ディレクトリを作成し、この**bin**ディレクトリをパスに追加するとよいでしょう。ほかのユーザーも使うスクリプトの場合には、**/usr/local/bin**など、新

しいプログラムを追加するのに適した場所にスクリプトを置きます。使用しているUNIXシステムでroot権限を持っていない場合には、システム管理者に依頼してファイルをコピーしてもらうことになります。ただし、その場合には、スクリプトの有用性や安全性についてシステム管理者を納得させなければなりません。ほかのユーザーが誤ってスクリプトを変更したりしないようにするには、スクリプトから書き込みアクセス権を削除します。管理者がスクリプトのオーナーとパーミッションを変更する場合には、次のようなコマンドを使うことになるでしょう。

```
# cp first.sh /usr/local/bin
# chown root /usr/local/bin/first.sh
# chgrp root /usr/local/bin/first.sh
# chmod 755 /usr/local/bin/first.sh
```

上の例では、必要なパーミッションがあらかじめ正確にわかっているので、パーミッションフラグを部分的に変更するのではなく、chmodの絶対モード指定を使っています。詳細については、chmodのマニュアルページを参照してください。

注意

UNIXでは、ディレクトリに対して書き込みパーミッションを持っているユーザーは、そのディレクトリにあるファイルを削除することができます。したがって、ほかのユーザーに消されたくないファイルは、rootしか書き込めないディレクトリに置く必要があります。

2.4 シェルの構文

簡単なシェルプログラムの例をすでに示したので、以下ではシェルの持つ強力なプログラミング機能について系統的に説明していきます。シェルは、学習の容易なプログラミング言語です。シェルを使えば、小さなプログラムの断片を対話的にテストし、期待どおりに動作するかどうかを確認したあと、より大きなスクリプトに組み込むことができます。機能の強化された新しいUNIXシェルを使えば、非常に大きな、構造化されたプログラムを作成することもできます。

ここでは、次のような項目を取り上げます。

- 変数：文字列、数値、環境、パラメータ
- 条件：シェルのブール型
- プログラム制御：if、elif、for、while、until、case
- リスト
- 関数
- シェルのビルトインコマンド

- コマンドの結果の取得
- ヒヤドキュメント

2.4.1 変数

通常、シェルでは変数を宣言せずに使用します。具体的には、変数に初期値を割り当てるなど、変数を最初に使用した時点で、その変数が作成されます。デフォルトでは、すべての変数は文字列型として扱われ、格納されます。これには、変数に数値が割り当てられている場合も含まれます。シェルと一部のユーティリティでは、値の操作が必要になったときに、"数値" 文字列を値に変換します。また、UNIXは大文字小文字を区別するシステムなので、シェルでも`foo`や`Foo`、`FOO`はいずれも別の変数として扱われます。

シェルでは、変数名の前に`$`文字を付けることで変数の内容を取得することができます。また、`echo`コマンドで変数の内容を出力することができます。変数を使う場合には、変数に値を割り当てる場合を除き、常に変数名の前に`$`を付ける必要があります。次に示す例では、コマンドラインで変数`salutation`にさまざまな値を設定しています。

```
$ salutation=Hello
$ echo $salutation
Hello
$ salutation="Yes Dear"
$ echo $salutation
Yes Dear
$ salutation=7+5
$ echo $salutation
7+5
```

注意

> 空白を含む文字列は引用符で囲む必要があります。また、等号の両側に空白を入れてはいけません。

`read`コマンドを使うと、ユーザーからの入力を変数に割り当てることができます。`read`コマンドは、ユーザーからの入力を受け取る変数の名前を引数に取り、ユーザーがテキストを入力するまで待ちます。通常、ユーザーが**Return**キーを押した時点で入力が終了したと判断し、処理を続行します。

◆ クォート

先に進む前に、シェルで重要な意味を持つクォートについて説明しておきましょう。

通常、パラメータは"ホワイトスペース文字"、つまり空白、タブ、または改行文字で区切ら

れています。このため、ホワイトスペース文字自体を含むパラメータを使う場合には、パラメータをクォートする必要があります。

$salutation**などの変数が、クォートされた内側の部分でどのように扱われるかは、クォートの種類によって異なります。**$による変数表現を二重引用符で囲むと、その行が実行されるときに変数が値に置き換えられます。単一引用符で囲むと、置き換えは行われません。また、**$**記号の前に\を付けると、**$**記号が持つ特別な意味が打ち消されます。

一般に、文字列は二重引用符で囲みます。これは、変数にホワイトスペースが含まれる場合でも1つの変数として扱えるようにするためと、**$**が展開されるようにするためです。

例題 変数

クォートによって変数の出力がどのように変わるかを実際に確かめてみましょう。

```
#!/bin/sh

myvar="Hi there"

echo $myvar
echo "$myvar"
echo '$myvar'
echo \$myvar

echo Enter some text
read myvar

echo '$myvar' now equals $myvar

exit 0
```

このスクリプトの出力は次のようになります。

```
Hi there
Hi there
$myvar
$myvar
Enter some text
Hello World
$myvar now equals Hello World
```

解説

変数**myvar**が作成され、文字列**Hi there**が割り当てられます。**echo**コマンドによって変数の内容が出力され、**$**文字の付いた変数が値に展開されることがわかります。二重引用符を使った場合にも、同様に変数の内容が出力されます。しかし、単一引用符とバックスラッシュ(円記

号）を使った場合には、値への置き換えは行われません。**read**コマンドは、ユーザーが入力した文字列を取得します。

◆ 環境変数

シェルスクリプトの実行が開始されると、いくつかの変数は、あらかじめ環境に含まれている値で初期化されます。通常、これらの変数には大文字が使われます。これは、スクリプトの中でユーザーが定義する変数（シェル変数）に小文字が使われることが多く、これら2つの種類の変数を互いに区別できるようにするためです。自動的に作成される変数は、ユーザー設定がどのようになっているかによって異なります。マニュアルページには多くの変数がリストアップされていますが、このうち特に重要なものをまとめたのが次の表です。

表2.2　環境変数

環境変数	説明
`$HOME`	現在のユーザーのホームディレクトリ。
`$PATH`	コマンドを検索するディレクトリのリストをコロンで区切ったもの。
`$PS1`	コマンドプロンプト（通常は`$`）。
`$PS2`	二次プロンプト（さらに入力が必要な場合に表示されるプロンプト。通常は`>`）。
`$IFS`	内部フィールドセパレータ。シェルが入力を読み取るときに語の区切りとして使う文字のリスト。通常は、空白、タブ、改行文字。
`$0`	シェルスクリプトの名前。
`$#`	渡されたパラメータの数。
`$$`	シェルスクリプトのプロセスID。しばしば、スクリプト内で一意の作業用ファイル名を生成するのに使われる（`/tmp/junk_$$`など）。

メモ

異なる環境に置いたときにプログラムがどのように動作するかをチェックしたい場合には、**env**コマンドを使います。詳しい使い方については、**env**のマニュアルページを参照してください。サブシェルの中で環境変数を設定するには、**export**コマンドを使います。このコマンドについては、あとで取り上げます。

◆ パラメータ変数

パラメータを指定してスクリプトを呼び出した場合には、いくつか追加の変数が作成されます。また、パラメータを1つも指定しなかった場合でも、上の表に示した環境変数の`$#`は存在し、その値は0に設定されます。

次の表は、パラメータ変数の一覧です。

表2.3 パラメータ変数

パラメータ変数	説明
$1、$2、...	スクリプトに与えられたパラメータ。
$*	すべてのパラメータのリストを1つの変数で表したもの。各パラメータは環境変数IFSの最初の文字で区切られる。
$@	$*とほぼ同じ。ただし、環境変数IFSは使われない。

X/Open仕様に従えば、$*と$@の違いは次のようになります。

二重引用符に囲まれた部分でパラメータの展開が行われる場合、$*は、各パラメータをIFS（内部フィールドセパレータ）の最初の文字（IFSが設定されていない場合には空白文字）で区切り、その全体を1つの値として持つ単一のフィールドに展開されます。IFSに空文字列が設定されている場合には、IFSが設定されていない場合とは異なり、パラメータのそれぞれの値が連結されます。次の具体例を見てください。

```
$ IFS=''
$ set foo bar bam
$ echo "$@"
foo bar bam
$ echo "$*"
foobarbam
$ unset IFS
$ echo "$*"
foo bar bam
```

この結果からわかるように、二重引用符で囲まれた$@は、IFSの値とは関係なく、各定位置パラメータを独立したフィールドとして展開します。

echoコマンドを使うと変数の内容を表示できますが、同様に、readコマンドを使って変数に値を読み込むこともできます。

例題 パラメータと環境変数

次のスクリプトは、ごく簡単な変数操作の例です。スクリプトの内容を入力し終わったら、chmod a+x try_variablesを実行してスクリプトを実行可能にしてください。

```
#!/bin/sh

salutation="Hello"
echo $salutation
echo "The program $0 is now running"
echo "The second parameter was $2"
echo "The first parameter was $1"
echo "The parameter list was $*"
echo "The user's home directory is $HOME"
```

```
echo "Please enter a new greeting"
read salutation

echo $salutation
echo "The script is now complete"

exit 0
```

スクリプトの実行結果は次のようになります。

```
$ try_variables foo bar baz
Hello
The program try_variables is now running
The second parameter was bar
The first parameter was foo
The parameter list was foo bar baz
The user's home directory is /home/rick
Please enter a new greeting
Sire
Sire
The script is now complete
$
```

解説

このスクリプトでは、変数**salutation**を作成し、その内容を表示しています。次に、各種パラメータ変数と環境変数**$HOME**がすでに存在し、それぞれ適切に値が設定されていることを確認しています。

パラメータの置き換えについては、あとでさらに詳しく取り上げます。

2.4.2 条件

条件を調べ、その結果に応じて異なる処理を実行する機能は、どんなプログラム言語にも不可欠です。ここでは、まずシェルスクリプトで使用できる条件の構文要素を取り上げ、これらの要素を使った制御構造について解説したあと、条件の使い方を学びます。

シェルスクリプトでは、コマンドラインから呼び出した任意のコマンドの終了コードを調べることができます。これには、自分で作成したスクリプトも含まれます。したがって、独自にスクリプトを作成する場合にも、スクリプトの最後に必ず**exit**コマンドを記述することが大切です。

◆ test（[]）コマンド

実際のスクリプトでは、シェルで真偽を判定するための**[]**コマンド（**test**コマンド）を非常によく使います。ほとんどのシステムでは、**[]**コマンドと**test**コマンドはまったく同じものです。

なぜ[]というコマンドがあるのか不思議に思うかもしれませんが、スクリプトの中で[]を使うと、コマンドを簡潔に記述することができます。また、ほかのプログラミング言語との類似性が高まるという利点もあります。

注意

UNIXシェルによっては、これらのコマンドから外部プログラムを呼び出すようになっているものもありますが、比較的新しいシェルでは、コマンドがシェル本体の中に組み込まれる傾向にあります。シェルのビルトインコマンドについては、あとで詳しく取り上げます。
testコマンドは、シェルスクリプト以外ではあまり使われません。そのためでしょうか、シェルスクリプトを書いたことのないUNIXユーザーは、自分が作成した簡単なプログラムに**test**という名前を付けることが多いようです。**test**という名前のプログラムがうまく動作しなかったら、ほとんどの場合、シェルの**test**コマンドと干渉していることが原因です。同名の外部コマンドがシステムに存在しているかどうかを調べるには、**which**を使って**which test**のように実行します。該当するコマンドが存在すれば、**/bin/test**または**/usr/bin/test**のように表示されます。

さっそく、**test**コマンドを使って単純な条件を調べてみましょう。調べる条件は、ファイルが存在するかどうかです。使用するコマンドは、**test -f filename**です。スクリプトでは、次のように記述します。

```
if test -f fred.c
then
...
fi
```

次のように記述することもできます。

```
if [ -f fred.c ]
then
...
fi
```

条件付きコードが実行されるかどうかは、**test**コマンドの終了コード（条件を満たしたかどうか）によって決まります。

注意

チェックする条件と角かっこ[]の間には空白が必要です。左の角かっこ（[）は、条件と右の角かっこ（]）を引数に取るコマンドだと覚えておくとよいでしょう。

ifと同じ行に**then**を記述したい場合には、条件判定と**then**を分離するためにセミコロン（**;**）を使います。

```
if [ -f fred.c ]; then
...
fi
```

testコマンドで扱うことができる条件の種類は、3つに分類できます。次に示す表は、種類ごとに条件をまとめたものです。

表2.4 文字列比較

文字列比較	結果
`string`	文字列が空文字列でなければ真
`string1 = string2`	2つの文字列が等しければ真
`string1 != string2`	2つの文字列が等しくなければ真
`-n string`	文字列が`null`でなければ真
`-z string`	文字列が`null`(空文字列)であれば真

表2.5 算術比較

算術比較	結果
`expression1 -eq expression2`	2つの式が等しければ真
`expression1 -ne expression2`	2つの式が等しくなければ真
`expression1 -gt expression2`	式1が式2より大きければ真(`gt`はgreater thanの意)
`expression1 -ge expression2`	式1が式2と等しいか、またはそれより大きければ真(`ge`はgreater than or equal toの意)
`expression1 -lt expression2`	式1が式2より小さければ真(`lt`はless thanの意)
`expression1 -le expression2`	式1が式2と等しいか、またはそれより小さければ真(`le`はless than or equal toの意)
`! expression`	`!`は式の否定。したがって、式が偽ならば真、式が真ならば偽

表2.6 ファイルの条件

ファイルの条件	結果
`-d file`	ファイルがディレクトリならば真
`-e file`	ファイルが存在すれば真
`-f file`	ファイルが通常のファイルならば真
`-g file`	ファイルに`set-group-id`ビットがセットされていれば真
`-r file`	ファイルが読み取り可能ならば真
`-s file`	ファイルのサイズが0でなければ真
`-u file`	ファイルに`set-user-id`ビットがセットされていれば真
`-w file`	ファイルが書き込み可能ならば真
`-x file`	ファイルが実行可能ならば真

注意

> -eオプションは移植性がないという歴史的経緯があるので、一般には-fオプションのほうがよく使われます。
> 上の表の説明に、set-group-idビットとset-user-idビットという耳慣れない用語が出てきました。これらはそれぞれset-gidビット、set-uidビットと呼ばれることもあります。set-uidビットは、プログラムに(ユーザーではなく)オーナーのパーミッションを与えます。同様に、set-gidビットは、プログラムにグループのパーミッションを与えます。
> set-gidビットとset-uidビットは、シェルスクリプトでは意味を持ちません。

上の表のどの条件の場合にも、結果が真を返すには、まず該当するファイルが存在していなければなりません。また、表に示したオプションは、testコマンドのオプションのうちよく使われるものだけです。その他のオプションについては、マニュアルページを参照してください。bashではtestはビルトインコマンドになっているので、詳しいヘルプを表示するには、help testと入力します。testのオプションについては、あとでまた取り上げます。

条件そのものについての説明は以上で終わりです。次に、条件を使った制御構造について説明します。

2.4.3 制御構造

シェルではさまざまな制御構造を利用することができます。シェルの制御構造は、ほかのプログラミング言語のものとよく似ています。制御構造によっては、caseステートメントのように、より強力な機能を持つものもあります。その他の制御構造も、構文がやや異なる程度の違いがあるだけです。

メモ

> 以下の説明では、条件が満たされるとき、満たされる間、または満たされるまでに実行される一連のコマンドをstatementsと表記します。条件自体はconditionと表記します。

◆ if

ifステートメントは、コマンドの結果を調べ、結果に応じて一連のステートメントを実行します。

```
if condition
then
  statements
else
  statements
fi
```

例題 ifコマンドの使用

次のように、質問に対する回答に応じて処理を変える場合には、ifをよく使います。

```
#!/bin/sh

echo "Is it morning? Please answer yes or no"
read timeofday

if [ $timeofday = "yes" ]; then
   echo "Good morning"
else
   echo "Good afternoon"
fi

exit 0
```

スクリプトの実行結果は次のようになります。

```
Is it morning? Please answer yes or no
yes
Good morning
$
```

解説

このスクリプトでは、[]コマンドを使って変数timeofdayの内容を調べています。そして、ifコマンドによる評価結果に応じて異なるコードを実行します。

> **注意**
> ifの内側では、ホワイトスペースを挿入してステートメントをインデントしていますが、これは単にスクリプトを（人間にとって）読みやすくするためです。シェルは、追加されたホワイトスペースを無視します。

◆ elif

上に示した簡単なスクリプトには、残念ながらいくつか問題があります。まず、yes以外の答えはすべてnoの意味になってしまいます。これを防ぐには、ifコマンドのelseの部分が実行されるときに、もうひとつ条件を加えます。

例題 elifを使った条件の追加

上のスクリプトを修正し、ユーザーの回答がyesまたはno以外の場合にはエラーメッセージを表示するようにしましょう。それには、elseをelifに変え、条件をもうひとつ加えます。

```
#!/bin/sh

echo "Is it morning? Please answer yes or no"
read timeofday

if [ $timeofday = "yes" ]; then
   echo "Good morning"
elif [ $timeofday = "no" ]; then
   echo "Good afternoon"
else
   echo "Sorry, $timeofday not recognized. Enter yes or no"
   exit 1
fi

exit 0
```

解説

最初に作成したスクリプトとほとんど同じですが、今度は**elif**を使用し、**if**コマンドの結果が偽だったときには変数をもう一度チェックします。どちらの条件判定の結果も偽になった場合には、エラーメッセージを表示し、終了コード1でスクリプトを終了します。終了コードを返すことで、ほかのプログラムからこのスクリプトを呼び出した場合でも、処理が成功したかどうかをチェックできるようにしています。

◆ 変数に関する問題

elifを使うことで、最も大きな問題は回避することができます。しかし、まだ微妙な問題が残っています。修整後のスクリプトを実行し、質問が表示されたときに**Return**キーだけを押したらどうなるでしょうか。この場合には、次のようなエラーメッセージが表示されます。

```
[: =: unary operator expected
```

どこがいけないのでしょうか。問題は、最初の**if**句にあります。変数**timeofday**をテストする時点で、この変数に設定されている文字列は空です。したがって、変数が展開されたあとの**if**句は次のようになっています。

```
if [ = "yes" ]
```

これは正しい条件になっていません。このような状況になるのを防ぐには、次のように変数を引用符で囲みます。

```
if [ "$timeofday" = "yes" ]
```

これで、変数に値が設定されていなくても、次のように正しい条件として評価されるようになります。

```
if [ "" = "yes" ]
```

修正を加えたあとのスクリプトは、次のようになります。

```
#!/bin/sh

echo -n "Is it morning? Please answer yes or no:  "
read timeofday

if [ "$timeofday" = "yes" ]; then
   echo "Good morning"
elif [ "$timeofday" = "no" ]; then
   echo "Good afternoon"
else
   echo "Sorry, $timeofday not recognized. Enter yes or no"
   exit 1
fi

exit 0
```

このスクリプトでは、質問に対してなにも入力せず、**Return** キーだけを押しても問題は起きません。

メモ

echo コマンドで末尾の改行文字を出力したくない場合には、echo -n とします。echo コマンドのその他のオプションについては、マニュアルページを参照してください。

◆ for

`for` は、ある範囲の値に対して繰り返し操作を行う場合に使います。値には、任意の文字列の集合を指定できます。指定する値は、プログラムの中にあらかじめ記述しておくこともできますが、通常はシェルによるファイル名の展開結果などを利用します。

構文自体は単純です。

```
for variable in values
do
  statements
done
```

例題 固定文字列を使ったforループ

通常、値には文字列を使って次のように記述します。

```
#!/bin/sh

for foo in bar fud 43
do
  echo $foo
done

exit 0
```

出力は次のようになります。

```
bar
fud
43
```

最初の行で**for foo in bar fud 43**を**for foo in "bar fud 43"**のように変更したらどうなるか試してみてください。この方法は、空白を含む文字列を変数に代入するときに利用できます。

解説

このスクリプトでは、変数**foo**が作成され、**for**ループを回るたびに、変数に異なる値が割り当てられます。シェルでは、どの変数もデフォルトでは文字列として扱われるので、文字列**fud**だけでなく、文字列**43**のような値を使うこともできます。

すでに述べたように、**for**ループは、シェルによるファイル名の展開と組み合わせて使うほうが一般的です。つまり、文字列の値としてワイルドカードを使用し、実行時にシェルによって具体的な値に展開されるようにするのです。

この方法は、すでにこの章の最初のスクリプト例**first.sh**でも使いました。**first.sh**では、シェルの展開、つまり*が現在のディレクトリに存在するすべてのファイル名に展開されることを利用しています。該当する各ファイル名は、**for**ループの中で変数**$file**として使われます。次にもうひとつ、ワイルドカードの展開を使った例を示しましょう。

例題 ワイルドカードの展開を使ったforループ

1冊の本の中からいくつかの章を印刷したいとしましょう。ファイルは章ごとに分かれています。この場合、次のスクリプトを実行すると、第3章～第5章だけを印刷することができます。

```
#!/bin/sh

for file in $(ls chap[345].txt); do
  lpr $file
done
```

解説

このスクリプトでは、`$(command)`という構文を使っています。この構文についてはあとで詳しく説明しますが、`for`コマンドのパラメータリストは、`$()`で囲まれたコマンドの出力によって設定されます。

すでに触れたように、`[345]`は3、4、5という文字の集合を表すので、`$()`で囲まれたコマンドはシェルによって展開され、`ls chap3.txt chap4.txt chap5.txt`のようになります。コマンドの実行結果は3つのテキストファイルの名前（ファイルが存在する場合）になり、これが`for`ループのパラメータになります。

> **注意**
> シェルスクリプトでの変数の展開は、スクリプトの実行時に行われます。このため、変数を宣言している部分での構文エラーは、スクリプトを実行するまで検出されません。すでに示した例で、質問に対して**Return**キーだけを押すと、`[: =: unary operator expected`というエラーが発生したのも、実行時に展開された変数が空文字列になったためです。

◆ while

シェル変数はいずれもデフォルトでは文字列として扱われるので、`for`ループは一連の文字列を対象として繰り返し処理を行う場合には便利です。しかし、指定した回数だけコマンドを実行する場合など、`for`ループではすっきりと記述できないケースもあります。

たとえば、20個の値を処理する場合、`for`ループでは次のようにスクリプトが冗長になってしまいます。

```
#!/bin/sh

for foo in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
do
  echo "here we go again"
done
exit 0
```

また、ワイルドカードによる展開を利用するとしても、どれだけ処理を繰り返せばよいのかはっきりとわからない場合があります。このような場合に便利なのが`while`ループです。構文は次のとおりです。

```
while condition
do
  statements
done
```

たとえば、パスワードの入力を求める簡単なスクリプトを`while`ループを使って記述すると、次のようになります。

```
#!/bin/sh

echo "Enter password"
read trythis

while [ "$trythis" != "secret" ]; do
  echo "Sorry, try again"
  read trythis
done

exit 0
```

このスクリプトの出力は次のようになります。

```
Enter password
password
Sorry, try again
secret
$
```

もちろん、このような方法でパスワードの入力を求めるのは、セキュリティ上好ましくありません。これは単に`while`ステートメントの使い方を説明するためのスクリプトです。`do`と`done`に挟まれたステートメントは、条件が真の間、繰り返し実行されます。条件の部分では、`trythis`の値が`secret`と異なるかどうかを調べています。したがって、`$trythis`が`secret`と一致するまでは、ループが実行されます。条件が真になると、`done`よりあとに記述されたステートメントが実行されます。

例題 繰り返しの実行

`while`を算術演算の代入と組み合わせると、一定の回数だけコマンドを実行でき、`for`ループを使った場合よりスクリプトを簡潔に記述できます。

```
#!/bin/sh

foo=1

while [ "$foo" -le 20 ]
do
```

```
  echo "Here we go again"
  foo=$(($foo+1))
done

exit 0
```

注意

$(( ))はkshで追加された構文要素ですが、それ以来X/Open仕様にも取り入れられています。kshより古いシェルでは、代わりにexprを使います。exprについてはあとで取り上げますが、$(())に比べて処理は遅く、より多くのリソースを消費します。

解説

このスクリプトは、[]コマンドを使ってfooの値を20と比較し、fooの値が20と等しいか、より小さい場合にはwhileループを実行します。whileループの中では、(($foo+1))という構文を使ってかっこの内側の式の算術演算を実行し、ループを回るたびにfooの値を1つ増やします。

このスクリプトでは、fooに空文字列が設定されることはないので、変数を二重引用符で囲む必要はありませんが、こうした習慣をつけるほうが好ましいという観点から二重引用符を使っています。

◆ until

untilステートメントの構文は次のとおりです。

```
until condition
do
  statements
done
```

untilステートメントはwhileループによく似ていますが、条件のテストが逆になっています。つまり、whileループでは条件が真の間ループが実行されますが、untilでは条件が真になるまでループが実行されます。

untilステートメントを使うと、条件判定をわかりやすく記述することができます。また、なにかが起こるまでループを無限に繰り返したい場合にも便利です。たとえば、次のスクリプトは、引数に指定したユーザーがログインしたときにメッセージを表示します。

```
#!/bin/sh

until who | grep "$1" > /dev/null
do
  sleep 60
```

```
done

# Now ring the bell and announce the expected user.

echo -e \\a
echo "***** $1 has just logged in *****"

exit 0
```

◆ case

caseの構文は次のとおりです。

```
case variable in
  pattern [ | pattern] ...) statements;;
  pattern [ | pattern] ...) statements;;
  ...
esac
```

構文は少し込み入っていますが、caseを使うと、変数の内容をパターンと突き合わせ、その結果に応じて異なるステートメントを実行することができます。

複数のパターン指定と複数のステートメントの実行が可能なことから、caseはユーザーの入力を調べるのに適しています。さっそく具体的な例を示しましょう。以下では、この例を出発点としてパターンマッチを改善していきます（ケース1～3）。

例題 ケース1: ユーザーからの入力

少し前に示した、ユーザーからの入力内容に応じて異なるメッセージを表示するスクリプトをcaseを使って書き直し、予想外の入力に対しても対処できるようにします。

```
#!/bin/sh

echo "Is it morning? Please answer yes or no"
read timeofday

case "$timeofday" in
    "yes") echo "Good Morning";;
    "no" ) echo "Good Afternoon";;
    "y"  ) echo "Good Morning";;
    "n"  ) echo "Good Afternoon";;
    *    ) echo "Sorry, answer not recognised";;
esac

exit 0
```

解説

caseステートメントが実行されると、timeofdayの内容が読み取られ、パターンに指定した各文字列と比較されます。文字列と入力内容とが一致すると、該当するパターンに対して指定されたステートメントが実行され、caseコマンドはただちに終了します。

caseコマンドは、比較の対象として使う文字列に対して通常の展開を行います。したがって、文字列の一部だけを指定し、そのあとにワイルドカードの*を付けることもできます。*だけを指定した場合には、すべての文字列に一致します。このため、いくつかパターンを指定したあと、必ず最後に*だけのパターンも用意し、ほかのパターンに一致しなかった場合でもcaseステートメントでデフォルトの処理を実行できるようにします。パターンの一致はベストマッチではなく、ファーストマッチで行われるので、最初に一致したものが使われます。デフォルトとして指定する条件は、予期していない条件でもあるので、*を使うとスクリプトのデバッグにも役立ちます。

例題 ケース2: パターンをまとめる

caseを使った上のスクリプトは、複数のifを使ったスクリプトよりも明らかにエレガントです。しかし、パターンをまとめると、さらに簡潔に記述することができます。

```
#!/bin/sh

echo "Is it morning? Please answer yes or no"
read timeofday

case "$timeofday" in
    "yes" | "y" | "Yes" | "YES" ) echo "Good Morning";;
    "n"*  | "N"* )                echo "Good Afternoon";;
    * )                           echo "Sorry, answer not recognised";;
esac

exit 0
```

解説

このスクリプトでは、caseの各エントリで複数の文字列を使っています。このため、実行するステートメントごとに、指定されたすべてのパターンが検査されます。パターンをまとめて記述すると、スクリプトを短く、読みやすくすることができます。また、*の使い方を示すために、2番目のエントリで"n"*や"N"*を使っています。ただし、これでは予期しない入力に一致してしまう可能性もあります。たとえば、ユーザーがneverと入力した場合でも"n"*に一致するため、Good Afternoonと表示されてしまいます。なお、ワイルドカード*の展開は、引用符の中では行われないことに注意してください。

例題 ケース3: 複数のステートメントの実行

作成したスクリプトを再利用できるようにするには、デフォルトのパターンが使われたときに、ほかのパターンとは異なる終了コードを返す必要があります。この点を考慮して書き直したのが次のスクリプトです。

```
#!/bin/sh

echo "Is it morning? Please answer yes or no"
read timeofday

case "$timeofday" in
    "yes" | "y" | "Yes" | "YES" )
           echo "Good Morning"
           echo "Up bright and early this morning?"
           ;;
    [nN]* )
           echo "Good Afternoon"
           ;;
    *)
           echo "Sorry, answer not recognised"
           echo "Please answer yes or no"
           exit 1
           ;;
esac

exit 0
```

解説

このスクリプトでは、さまざまなパターンマッチングの使い方を示すために、ユーザーが「いいえ」と答えた場合のパターンを変えています。また、caseステートメントの各パターンで、複数のステートメントを実行しています。このスクリプトでは、パターンを指定するときに、一致する範囲の狭いパターンを最初に、範囲の広いパターンを最後に記述していることに注意してください。これは、caseステートメントでは、ベストマッチではなくファーストマッチ、つまり最初に一致した部分が実行されるためです。もし*)を最初に記述したとすると、入力した内容にかかわらず、常に*)とマッチしてしまうことになります。

> **注意**
>
> esacの直前の;;は省略可能です。Cプログラミングでbreakを省くのはよい習慣とはいえませんが、シェルスクリプトの場合、最後のパターンがデフォルトのパターンならば、ほかにパターンが調べられることはないので、esacの前の;;を省いてもまったく問題ありません。

caseのマッチングは、次のようにパターンを指定することでさらに強力になります。

```
[yY] | [Yy][Ee][Ss] )
```

このパターン指定を使うと、一致する文字をさらに限定できるので、さまざまな回答に対応しつつ、ワイルドカード*を使った場合よりも細かい制御が可能になります。

◆ リスト

場合によっては、複数のコマンドをつなげたいことがあります。たとえば、あるステートメントを実行するために、複数の条件を満たす必要がある場合などです。

```
if [ -f this_file ]; then
    if [ -f that_file ]; then
        if [ -f the_other_file ]; then
            echo "All files present, and correct"
        fi
    fi
fi
```

あるいは、複数の条件のうち少なくとも1つを満たしたときに、なんらかの処理を実行したい場合もあります。たとえば、次の例のような場合です。

```
if [ -f this_file ]; then
   foo="True"
elif [ -f that_file ]; then
   foo="True"
elif [ -f the_other_file ]; then
   foo="True"
else
   foo="False"
fi
if [ "$foo"= "True"]; then
   echo "One of the files exists"
if
```

もちろん、ifステートメントを複数使ってもよいのですが、上の例を見てもわかるように、無駄の多い煩雑なスクリプトになってしまいます。実は、シェルには、コマンドのリストを使うための特別な構文要素があります。それはANDリストとORリストです。ANDリストとORリストは一緒に使われることも多いのですが、ここでは分けて説明します。

ANDリスト

ANDリストを使うと、一連のコマンドを実行することができます。ただし、リスト内の次のコマンドが実行されるのは、その直前までのすべてのコマンドの実行が成功した場合だけです。構文は次のとおりです。

```
statement1 && statement2 && statement3 && ...
```

ANDリストでは、一番左のステートメントから実行され、実行されたステートメントが`true`を返した場合には、その次のステートメントが実行されます。そして、ステートメントが`false`を返すまでこれが繰り返されます。実行されたステートメントが`false`を返した場合、それ以降のステートメントは実行されません。`&&`は、その前のコマンドの結果を調べます。

各ステートメントは独立して実行されるので、すぐあとに示す例のように、1つのリストに多くの異なるコマンドを混在させることができます。ANDリストに含まれるすべてのコマンドの実行が成功した場合には、ANDリスト全体の実行が正常に終了したことになります。それ以外の場合、実行は失敗したことになります。

例題 ANDリスト

次のスクリプトでは、`touch file_one`によって、ファイルが存在していなければ作成し、`file_two`を削除します。次に、ANDリストで各ファイルの存在を調べ、その間に`echo`コマンドで短いテキストを出力します。

```
#!/bin/sh

touch file_one
rm -f file_two

if [ -f file_one ] && echo "hello" && [ -f file_two ] && echo "there"
then
    echo "in if"
else
    echo "in else"
fi

exit 0
```

スクリプトを実行すると、次のように出力されます。

```
hello
in else
```

解説

`touch`コマンドと`rm`コマンドによって、スクリプトで使うファイルの状態をあらかじめ設定しておきます。`&&`リストでは、まず`[ -f file_one ]`ステートメントが実行されます。ファイルは存在するので、このステートメントの実行は成功し、次の`echo`コマンドが実行されます。これも成功するので（`echo`は常に`true`を返します）、その次の`[ -f file_two ]`が実行されま

す。今度はファイルが存在しないので、実行は失敗します。その次のechoステートメントは、前のコマンドの実行が失敗しているため、実行されません。&&リストの結果は、リストに含まれる1つのコマンドの実行が失敗しているため、全体としてfalseとなります。このため、ifステートメントのelse条件の部分が実行されます。

ORリスト

ORリストを使うと、リスト内の1つのコマンドの実行が成功するまで、一連のコマンドを実行することができます。リスト内のコマンドの実行が成功した場合、それ以降のコマンドは実行されません。構文は次のとおりです。

```
statement1 || statement2 || statement3 || ...
```

ORリストでは、一番左のステートメントから実行され、実行されたステートメントがfalseを返した場合には、その次のステートメントが実行されます。そして、ステートメントがtrueを返すまでこれが繰り返されます。実行されたステートメントがtrueを返した場合、それ以降のステートメントは実行されません。

||リストは&&リストによく似ていますが、前のステートメントの実行が失敗した場合に次のステートメントを実行する点が異なります。

例題 ORリスト

上のANDリストの例をコピーし、次のように修正します。

```
#!/bin/sh

rm -f file_one

if [ -f file_one ] || echo "hello" || echo "there"
then
    echo "in if"
else
    echo "in else"
fi

exit 0
```

スクリプトを実行すると、次のように出力されます。

```
hello
in if
```

解説

最初の行では、スクリプトで使うファイルの設定を行っています。ORリストの最初のコマンド`[ -f file_one ]`の実行は、ファイルが存在しないため失敗します。次に、echoコマンドが実行されます。echoコマンドは`true`を返すので、`||`リスト内のそれ以降のコマンドは実行されません。`||`リスト内の1つのコマンド（echoコマンド）が`true`を返しているので、結果として`if`の部分が実行されます。

ANDリストとORリストのどちらの場合にも、最後に実行されたステートメントの結果がリスト全体の結果となります。

これらのリストは、Cで複数の条件をテストする場合と同じように動作し、結果が出た時点でそれ以上のステートメントは実行されません。つまり、必要最小限のステートメントだけが実行され、結果に影響しないステートメントは実行されないのです。こうした評価の方法を、一般に"ショートサーキット評価"と呼びます。

ANDリストとORリストを組み合わせると、論理式を思いどおりに組み立てることができます。たとえば、次の例を見てください。

```
[ -f file_one ] && command for true || command for false
```

この例では、先頭の条件が真であれば、2つのコマンドのうち最初のコマンドが実行され、そうでない場合には2番目のコマンドが実行されます。この記述方法は便利なので、ぜひ覚えておいてください。

◆ ステートメントブロック

ANDリストやORリストのコマンドのように、1つのステートメントしか記述できないところで、複数のステートメントを実行するにはどうすればよいのでしょうか。このような場合には、中かっこ（`{}`）を使ってステートメントブロックを作成します。次に示すのは、この章のあとの部分で作成するアプリケーションで使われている例です。

```
get_confirm && {
  grep -v "$cdcatnum"$tracks_file > $temp_file
  mv $temp_file $tracks_file
  cat $temp_file > $tracks_file
  echo
  add_record_tracks
}
```

2.4.4 関数

シェルでは関数を定義することができます。関数を使うと、大きなシェルスクリプトを書く場合でも、コードを構造化することができます。

メモ

関数を使う代わりに、大きなスクリプトを複数の小さなスクリプトに分割し、その中である特定の作業を実行する方法もあります。ただし、この方法には欠点もあります。まず、スクリプトの中から別のスクリプトを呼び出すと、関数を実行する場合と比べて、処理速度が低下します。また、スクリプトの実行結果をどうやって受け取るかにも工夫が必要になり、小さなスクリプトをたくさん用意しなければならなくなることがあります。なお、関数を使ったり、別のスクリプトを呼び出したりする場合には、スクリプトの中で実行単位として明確に意味を持つ独立性の高い部分はどこかをよく考える必要があります。
大きなプログラムをシェルで記述してよいのかどうか、疑問を持つ読者もいるかもしれません。しかし、FSFの**autoconf**やUNIXのパッケージインストールプログラムなど、多くのプログラムはシェルスクリプトで記述されています。基本的な機能を持つシェルは、どんなUNIXシステムにも必ず用意されています。そもそも、ほとんどのUNIXシステムでは、**/bin/sh**がなければ、ログインすることはおろかブートすることすら不可能です。ですから安心してシェルをプログラミングに利用することができます。

シェルで関数を定義するには、関数名に続けて空のかっこ（**()**）を記述し、そのあとに中かっこ（**{}**）で囲んで、実行するステートメントを記述します。

```
function_name () {
  statements
}
```

例題 単純な関数

まず、ごく簡単な関数を書いてみましょう。

```
#!/bin/sh

foo() {
    echo "Function foo is executing"
}

echo "script starting"
foo
echo "script ended"

exit 0
```

スクリプトの実行結果は次のようになります。

```
script starting
Function foo is executing
script ending
```

解説

スクリプトは通常どおり上から順に実行されます。しかし、**foo ()** {という構文要素は**foo**という名前の関数の定義なので、シェルはこれ以降**foo**を関数として認識することにして、関数定義の終わりに対応する}までを記憶します。その後、**foo**とだけ記述された行に出会うと、あらかじめ定義されていた関数が実行されます。関数の実行が終わると、**foo**を呼び出した次の行からスクリプトの実行を続けます。

関数は、呼び出す前に定義しておく必要があります。これはPascalでの関数定義の方法に似ていますが、シェルでは事前の宣言がない点が異なります。ただし、シェルスクリプトは常に先頭の行から実行されるので、これは問題になりません。複数の関数を使うときには、いずれかの関数を呼び出す前にすべての関数をあらかじめ定義しておけば、呼び出した関数が定義されていないといったエラーの発生を未然に防ぐことができます。

関数が呼び出されると、スクリプトの定位置パラメータ**$***、**$@**、**$#**、**$1**、**$2**などは関数のパラメータに置き換えられます。関数に渡された引数は、これらのパラメータから読み取られます。関数の実行が終了すると、パラメータはもとの値に戻されます。

注意

> 古いシェルの中には、関数の実行後に定位置パラメータの値を復元しないものがあります。移植性の高いスクリプトを記述したい場合には、この点に注意してください。

関数の中で**return**コマンドを使うと、関数から数値を返すことができます。一方、関数から文字列を返すには、関数の中で変数に文字列を格納します。関数の実行後、この変数を参照すれば、設定された文字列を取得できます。

キーワード**local**を使うと、シェルの関数の中でローカル変数を宣言できます。**local**を指定して宣言した変数のスコープ（有効範囲）は、関数の中だけです。これ以外の変数のスコープは基本的にグローバルなので、関数の中でほかのシェル変数にアクセスすることもできます。ローカル変数がグローバル変数と同じ名前を持つ場合には、関数内部でのみローカル変数がグローバル変数を上書きします。たとえば、関数の例として出した上のスクリプトを次のように修正すると、ローカル変数とグローバル変数がどのように扱われるかを確かめることができます。

```
#!/bin/sh

sample_text="global variable"

foo() {
    local sample_text="local variable"

    echo "Function foo is executing"
    echo $sample_text
}

echo "script starting"
echo $sample_text

foo

echo "script ended"
echo $sample_text

exit 0
```

上の関数には戻り値を指定するための`return`コマンドがないので、最後に実行されたコマンドの終了コードが関数の結果として返されます。

次の`my_name`スクリプトでは、関数にパラメータを渡す方法と、関数から`true`または`false`を返す方法を示しています。

例題 関数からの戻り値

1 シェルのヘッダーのすぐあとで、関数`yes_or_no`を定義します。

```
#!/bin/sh

yes_or_no() {
  echo "Parameters are $*"
  while true
  do
    echo "Enter yes or no"
    read x
    case "$x" in
      y | yes ) return 0;;
      n | no )  return 1;;
      * )       echo "Answer yes or no"
    esac
  done
}
```

2 次に、プログラム本体を記述します。

```
echo "Original parameters are $*"

if yes_or_no "Is your name $1"
then
  echo "Hi $1"
else
  echo "Never mind"
fi

exit 0
```

スクリプトを実行すると、次のような結果になります。

```
$ my_name Rick and Neil
Original parameters are Rick and Neil
Parameters are Is your name Rick
Enter yes or no
no
Never mind
```

解説

スクリプトを実行すると、関数`yes_or_no`が定義されます。この部分は関数の定義なので、まだ実行されません。次に、`if`ステートメントで関数`yes_or_no`が実行されます。このとき、`$1`がスクリプト自体の最初のパラメータ（`Rick`）で置き換えられたあと、行末までの部分がパラメータとして関数に渡されます。関数では、定位置パラメータ`$1`、`$2`などで渡された引数を利用して必要な処理を行い、呼び出し元に値を返します。その後、関数からの戻り値が`if`によって評価され、適切なステートメントが実行されます。

以上のように、シェルにはさまざまな制御構造と条件ステートメントが用意されています。次は、シェルに組み込まれているコマンド（ビルトインコマンド）について説明し、コンパイラを使わなくても十分実用的なプログラムを作成できることを示します。

2.4.5 コマンド

シェルスクリプトから実行できるコマンドには2つの種類があります。ひとつは、コマンドプロンプトからも実行可能な“通常の”コマンドで、もうひとつはすでに簡単に触れたビルトインコマンドです。ビルトインコマンドはシェルの内部に実装されており、外部プログラムとして呼び出すことはできません。ただし、ほとんどの内部コマンドは、スタンドアロンのプログラムとしても用意されています（これはPOSIX仕様の一部になっています）。一般的にいって、内部コマンドのほうが効率よく実行できる点を除けば、内部コマンドか外部コマンドかの違いは重要ではありません。

メモ

UNIXでは、いくつかのコマンドやファイルを1つのプログラムで済ませていることがあります。たとえば、ファイルの圧縮と解凍を行う`gzip`、`gunzip`というプログラムを`ls -l`で表示させてみてください。`gunzip`は`gzip`へのシンボリックリンクになっています。多くのUNIXシステムでは、このように`ln`（リンク）コマンドを使って、1つのファイルに複数の名前を付けています。この種のコマンドでは、最初の引数（UNIXでは実行されたコマンドの名前が入る）を見て、行うべき処理を決定しています。

ここでは、内部コマンドと外部コマンドを含めて、スクリプトの作成に必要な主要なコマンドしか紹介しません。UNIXユーザーであれば、コマンドプロンプトから実行可能な多くのコマンドをすでに知っていることでしょう。これらのコマンドは、ここで紹介するビルトインコマンドに加えて、シェルスクリプトの中で使うことができます。

◆ break

`break`は、条件を満たす前に、`for`、`while`、または`until`ループから抜けるときに使います。

```
#!/bin/sh

rm -rf fred*
echo > fred1
echo > fred2
mkdir fred3
echo > fred4

for file in fred*
do
  if [ -d "$file" ]; then
    break;
  fi
done

echo first directory fred was $file

exit 0
```

`break`には、抜けるループの数をパラメータとして指定することができます。これは機能としては便利ですが、マニュアルに詳しい説明がないせいか、ほとんど使われていません。また、実際問題としてスクリプトの構造がわかりにくくなるので、あまり使わないほうがよいでしょう。

◆ :コマンド

コロン（`:`）コマンドは、ヌルコマンドです。`:`は`true`コマンドの代わりに使用でき、`:`を使うことで条件を簡潔に記述できる場合があります。`:`はビルトインコマンドなので、`true`よりも実行効率はよいのですが、多用するとスクリプトがわかりにくくなるのが難点です。

多くの場合、:コマンドは、whileループの条件として使われます。while :は、while trueと同じで、無限ループを形成します。

:は、条件に応じて変数の設定を行う場合にも便利です。たとえば、次のように使います。

```
: ${var:=value}
```

もし最初の:がなければ、$varは、シェルによってコマンドとして評価されることになります。


注意

> 一部の古いシェルスクリプトでは、行頭にコロンを置いてコメントを記述していることがあります。しかし、新しいシェルを使う場合には、コメント行は#で始めるべきです。また、そのほうがスクリプトを効率よく実行できます。

```
#!/bin/sh

rm -f fred
if [ -f fred ]; then
  :
else
  echo file fred did not exist
fi

exit 0
```

◆ continue

同名のCのステートメントと同様、このコマンドは一番内側のfor、while、またはuntilループの始まりに戻り、次の繰り返しを始めます。このとき、ループ変数にはリストの次の値がセットされます。

```
#!/bin/sh

rm -rf fred*
echo > fred1
echo > fred2
mkdir fred3
echo > fred4

for file in fred*
do
  if [ -d "$file" ]; then
    continue
  fi
  echo file is $file
done

exit 0
```

continueは、パラメータを1つ取ることができます。このパラメータは、何番目のループに戻るかを指定するためのものです。このパラメータを使えば、ループがいくつかネストされている場合でも、ループをまとめて飛び越えることができます。ただし、このパラメータはほとんど使われません。次に例を示します。

```
for x in 1 2 3
do
  echo before $x
  continue 1
  echo after $x
done
```

このスクリプトの出力は次のようになります。

```
before 1
before 2
before 3
```

◆ .コマンド

ドット（.）コマンドは、現在のシェルでコマンドを実行します。たとえば、次のように使います。

```
. shell_script
```

通常、スクリプトから外部のコマンドやスクリプトを実行すると、新しい環境（**サブシェル**）が作成され、この新しい環境でコマンドが実行されます。実行が終わると、作成された新しい環境は破棄され、終了コードだけが親シェルに返されます。しかし、**source**コマンドやドットコマンド（どちらも同じです）を使うと、スクリプトを呼び出したシェルと同じシェルでコマンドを実行することができます。

つまり、スクリプトから呼び出したコマンドで環境変数を変更しても、その内容は失われてしまいますが、ドットコマンドを使えば、実行するコマンドの中で現在の環境を変更することができます。これは、ほかのコマンドを実行するための環境設定を行うラッパーとしてスクリプトを作成するときに便利です。たとえば、同時に複数のプロジェクトを進めていて、古いプログラムの保守のために以前のバージョンのコンパイラを使わなければならない場合があるとします。このような場合、ドットコマンドを使えば、古いコンパイラ用のパラメータを簡単に設定することができます。

シェルスクリプトでは、ドットコマンドはCやC++の**#include**ディレクティブのように動作します。ただし、スクリプトを文字どおりにインクルードするのではなく、現在のコンテキストの中で、指定されたコマンドを実行します。このことを利用すると、変数や関数の定義をスクリプトに簡単に取り込むことができます。

次に示す例では、コマンドラインでドットコマンドを使っていますが、スクリプトの中でも使用できることを覚えておいてください。

例題 .コマンド

1 2つの異なる開発環境用の設定を含む2つのファイルがあるとします。一方は、古い従来のコマンドの環境を設定するための`classic_set`ファイルです。

```
#!/bin/sh

version=classic
PATH=/usr/local/old_bin:/usr/bin:/bin:.
PS1="classic> "
```

2 一方、新しいコマンドの環境を設定するには、次の`latest_set`ファイルを使います。

```
#!/bin/sh

version=latest
PATH=/usr/local/new_bin:/usr/bin:/bin:.
PS1="1999> "
```

ドットコマンドとこれら2つのファイルを組み合わせると、次のように目的の環境を簡単に設定できます。

```
$ . classic_set
classic> echo $version
classic
classic> . latest_set
1999> echo $version
latest
1999>
```

◆ echo

X/Openでは、新しいシェルの場合には`printf`コマンドを使うことを強く推奨しています。ただし、本書では、最後に改行文字を伴う文字列を出力するために、これまで`echo`コマンドを使ってきました。

よく問題になるのは、最後の改行文字を出力したくない場合です。残念なことに、UNIXのバージョンによってその解決方法は異なっており、移植性の高い方法でこれを実現する手段はありません。通常は、次のような方法を使います。

```
echo -n "string to output"
```

しかし、次のような方法を使う場合も少なくありません。

```
echo -e "string to output\c"
```

注意

> 2番目の`echo -e`では、タブを表す`\t`やキャリッジリターンを表す`\r`など、バックスラッシュでエスケープされた文字の解釈が有効になります。通常、このオプションはデフォルトで有効になっています。詳細については、マニュアルページで確認してください。

◆ eval

`eval`コマンドを使うと、引数を評価することができます。このコマンドはシェルに組み込まれており、通常は独立したコマンドとしては存在しません。ここでは、X/Open仕様そのものに含まれている簡単な例を引きます。

```
foo=10
x=foo
y='$'$x
echo $y
```

これを実行すると、`$foo`と出力されます。ここで、`eval`を使って次のように記述します。

```
foo=10
x=foo
eval y='$'$x
echo $y
```

今度は、`10`が出力されます。この場合、`eval`はもうひとつ`$`がある場合と同じような動作をするので、変数の値のそのまた値を取得できるのです。

`eval`コマンドは、コードを生成して即座に実行できるので非常に便利です。`eval`コマンドを使うと、スクリプトのデバッグはやや難しくなりますが、ほかの手段では実現できないことが可能になる点は大きな魅力です。

◆ exec

`exec`コマンドには2つの使い方があります。まず、よく使われるのは、現在のシェルを別のプログラムで置き換えるという使い方です。次の例を見てください。

```
exec wall "Thanks for all the fish."
```

シェルスクリプトにこの行があると、現在のシェルが`wall`コマンドによって置き換えられま

す。execが処理されたあと、スクリプトの中でexecよりあとにある行は実行されません。スクリプトを実行していたシェルが存在しなくなるからです。

execのもうひとつの使い方は、現在のファイルデスクリプタを変更するという使い方です。

```
exec 3< afile
```

このコマンドは、ファイルデスクリプタ3を開き、これをファイルafileからの読み取りに使います。ただし、このような使い方は実際にはまれです。

◆ exit n

exitコマンドは、終了コードnでスクリプトを終了します。対話的シェルのコマンドプロンプトでexitコマンドを使うと、ログアウトすることになります。スクリプトで終了コードを指定せずに終了した場合には、スクリプト内で最後に実行されたコマンドの終了コードがスクリプトの戻り値として使われます。ただし、通常、スクリプトでは常に明示的に終了コードを返すようにしておきます。

シェルスクリプトプログラミングでは、終了コード0は成功を表します。終了コード1から125までは、スクリプトで使用できるエラーコードです。これ以外の値は、次の表のように、あらかじめ意味が予約されています。

表2.7 スクリプトの終了コード

終了コード	説明
126	ファイルを実行できなかった
127	コマンドが見つからなかった
128以上	シグナルを受け取った

CやC++のプログラマは、0が成功を表すことに違和感を覚えるかもしれません。しかしスクリプトでは、エラーコード用のグローバル変数を用意しなくても、125のユーザー定義エラーコードを使用することができます。これは大きな利点です。

次に示すのは、.profileという名前のファイルが現在のディレクトリに存在すれば、終了コード0を返す簡単な例です。

```
#!/bin/sh

if [ -f .profile ]; then
    exit 0
fi

exit 1
```

できるだけ凝って簡潔なスクリプトを作成したい場合には、すでに取り上げたANDリストと

ORリストを使って次のように記述することができます。

```
[ -f .profile ] && exit 0 || exit 1
```

◆ export

exportコマンドは、パラメータに指定された変数をサブシェルでも利用できるようにします。デフォルトでは、あるシェル内で作成された変数は、そのシェルから呼び出された（サブ）シェルでは利用できません。exportコマンドは、指定されたパラメータから環境変数を作成し、現在のプログラムから呼び出したほかのスクリプトやプログラムからも変数を利用できるようにします。技術的に正確にいえば、exportコマンドによってエクスポートされた変数は、そのシェルから派生したすべての子プロセスの環境変数となります。次に、export1とexport2という2つのスクリプトを使って具体的な例を示します。

例題 変数のエクスポート

1 最初にexport2スクリプトの内容を示します。

```
#!/bin/sh

echo "$foo"
echo "$bar"
```

2 次はexport1スクリプトです。スクリプトの最後でexport2を呼び出します。

```
#!/bin/sh

foo="The first meta-syntactic variable"
export bar="The second meta-syntactic variable"

export2
```

スクリプトの実行結果は次のようになります。

```
$ export1

The second meta-syntactic variable
$
```

解説

スクリプトを実行した直後に空行が出力されているのは、export2では変数fooを利用できず、$fooの評価結果が空文字列になるからです。空文字列をechoで表示すると、改行文字だけ

が出力されます。

変数がシェルからエクスポートされると、そのシェルから呼び出された任意のスクリプトに変数がエクスポートされ、さらにこれらのスクリプトから呼び出されたシェルにも変数がエクスポートされます。以下、同様にエクスポートが行われます。スクリプト**export2**がほかのスクリプトを呼び出した場合には、呼び出されたスクリプトでも**bar**の値を利用することができます。

> **メモ**
> **set -a**コマンドと**set -allexport**コマンドは、それ以後のすべての変数をエクスポートします。

◆ expr

exprコマンドは、指定された引数を式として評価します。**expr**コマンドは、次のような単純な算術演算でよく使われます。

```
x=`expr $x + 1`
```

> **注意**
> `で囲まれている部分は、**expr $x + 1**というコマンドの実行結果に置き換えられます。コマンドの置換については、あとで詳しく取り上げます。

exprは、さまざまな式の評価を行うことができる強力なコマンドです。次に示すのは、**expr**コマンドで指定できる主な式の一覧です。

表2.8 exprコマンドで評価できる式

式の評価	説明
expr1 | expr2	expr1が0でないならexpr1、それ以外の場合はexpr2
expr1 & expr2	どちらの式も0なら0、それ以外の場合はexpr1
expr1 = expr2	等しい
expr1 > expr2	より大きい
expr1 >= expr2	等しいか、またはより大きい
expr1 < expr2	より小さい
expr1 <= expr2	等しいか、またはより小さい
expr1 != expr2	等しくない
expr1 + expr2	加算
expr1 - expr2	減算
expr1 * expr2	乗算
expr1 / expr2	整数除算
expr1 % expr2	整数剰余

新しいシェルでは、**expr**よりも**$((...))**構文を使ったほうが効率的です。**$((...))**構文についてはまたあとで取り上げます。

◆ printf

printfコマンドは、比較的新しいシェルでのみ利用できます。X/Openでは、書式付き出力を生成する場合には**echo**ではなく、**printf**を使用すべきとしています。

printfコマンドの構文は次のとおりです。

```
printf "format string" parameter1 parameter2 ...
```

書式文字列（上の構文中の**format string**）は、CやC++で使われるものとよく似ていますが、一部に制限があります。最も大きな制限は、浮動小数点がサポートされていない点です。これは、シェルではすべての演算が整数を対象に行われるためです。書式文字列には、リテラル文字、エスケープシーケンス、変換指定子の任意の組み合わせを指定できます。書式文字列中の%と\を除くすべての文字は、そのまま出力に含まれます。

次の表は、サポートされているエスケープシーケンスの一覧です。

表2.9 printfコマンドで使うエスケープシーケンス

エスケープシーケンス	説明
\\	バックスラッシュ文字
\a	警告（ベルまたはビープ音を鳴らす）
\b	バックスペース文字
\f	フォームフィード文字
\n	改行文字
\r	キャリッジリターン
\t	タブ文字
\v	垂直タブ文字
\ooo	8進数で値oooを持つ1文字

変換指定子にはさまざまなものがあるので、よく使うものだけを下の表にまとめます。詳細についてはマニュアルを参照してください。変換指定子は、1つの%文字と、それに続く変換文字によって表されます。

表2.10 printfコマンドで使う変換指定子

変換指定子	説明
d	10進数を出力
c	文字を出力
s	文字列を出力
%	%文字そのものを出力

書式文字列は、それ以降のパラメータを解釈して結果を出力するために使われます。次に例を示します。

```
$ printf "%s\n" hello
hello
$ printf "% %d\t%s" "Hi There" 15 people
Hi There 15  people
```

2番目の例では、空白を含む文字列Hi Thereを1つのパラメータとして渡すために、二重引用符を使っています。

◆ return

returnコマンドは、関数から戻るために使います。このコマンドについては、関数を取り上げたときにすでに簡単に触れました。**return**は1つの数値パラメータを取ります。この値は、関数を呼び出したスクリプトで利用することができます。**return**にパラメータを指定しなかった場合には、最後に実行されたコマンドの終了コードが返されます。

◆ set

setコマンドは、シェルのパラメータ変数を設定します。**set**コマンドを使うと、空白で区切られた複数の値を出力するコマンドから、目的のフィールドを簡単に取得することができます。

たとえば、シェルスクリプトの中で今月の月の名前を使いたいとしましょう。システムには、月の名前を文字列として出力する**date**コマンドが用意されています。しかし、曜日や時間なども一緒に出力されるので、これらのフィールドの中から月の名前だけを切り出す必要があります。そのためには、**date**コマンドを実行して結果を返す構文要素**$(...)**と、**set**コマンドを組み合わせて使います。**date**コマンドは月の名前を2番目のパラメータとして出力するので、次のようにスクリプトを記述します。

```
#!/bin/sh

echo The date today date is $(date)
set $(date)
echo The month is $2

exit 0
```

このスクリプトは、**date**コマンドからの出力をパラメータリストに設定し、定位置パラメータ**$2**を使って月の名前を取り出します。

setコマンドは、シェルの実行方法を制御するときにも利用できます。よく使われるのは、現在実行しているコマンドを表示させる**set -x**です。

メモ

setコマンドとそのオプションについては、あとでデバッグについて説明するときにもう一度取り上げます。

◆ shift

shiftコマンドは、パラメータ変数の$2を$1に、$3を$2にというように、すべてのパラメータ変数を1つずつずらします。$1に設定されていた以前の値は破棄されますが、$0の内容は変更されません。shiftコマンドを実行すると、その他の変数$*、$@、$#も、新しいパラメータ変数に対応する形で変化します。

shiftは、パラメータを順に検査する場合に便利です。また、10以上のパラメータを使うスクリプトでは、10番目以降のパラメータにアクセスするためにshiftを使う必要があります。

次に示すのは、与えられたすべての定位置パラメータを出力するスクリプトです。

```
#!/bin/sh

while [ "$1" != "" ]; do
  echo "$1"
  shift
done

exit 0
```

◆ trap

trapコマンドは、シグナルを受け取ったときの動作を指定します（シグナルについては、あとの章で詳しく説明します）。一般に、trapは、スクリプトが中断されたときの後始末をするために使います。従来、シェルではシグナルを番号で指定するのが一般的でしたが、新しく作成するスクリプトでは、インクルードファイルsignal.hに列挙されている名前から先頭のSIGを取り去ったものを使ったほうがよいとされています。どのようなシグナルがあるかを調べるには、trap -lと入力します。

メモ

シグナルについてあまりよく知らない読者のために解説しておくと、シグナルとは、非同期的にプログラムに送られるイベントのことです。通常、シグナルを送られたプログラムは終了します。

trapコマンドには、シグナルを受け取ったときに実行する動作と、トラップ（捕捉）するシグナルの名前を指定します。シグナルの名前は複数指定することができます。

```
trap command signal
```

通常、スクリプトは上から順に解釈されるので、trapコマンドは、対象としたい部分よりも前に記述する必要があります。

トラップの条件をデフォルトに戻すには、動作に-を指定します。シグナルを無視するには、動作を指定する文字列に空文字列''を指定します。パラメータを指定せずにtrapコマンドを実行すると、現在のトラップ指定と動作の一覧が表示されます。

次の表は、X/Open規格に記載されているものの中から、trapコマンドでよく使う重要なシグナルをまとめたものです。

表2.11 trapコマンドで使うシグナル

シグナル	説明
HUP (1)	ハングアップ。端末がオフラインになったり、ユーザーがログアウトしたときに送られる。
INT (2)	インタラプト(割り込み)。通常、**Ctrl-C**を押したときに送られる。
QUIT (3)	クイット。通常、**Ctrl-**を押したときに送られる。
ABRT (6)	アボート。重大な実行エラーが発生した場合などに送られる。
ALRM (14)	アラーム。通常、タイウアウトを処理するのに使われる。
TERM (15)	終了。通常、システムがシャットダウンするときに送られる。

例題 シグナルのトラップ

次のスクリプトは、シグナルの扱い方を示す簡単な例です。

```
#!/bin/sh

trap 'rm -f /tmp/my_tmp_file_$$' INT
echo creating file /tmp/my_tmp_file_$$
date > /tmp/my_tmp_file_$$

echo "press interrupt (Ctrl-C) to interrupt ...."
while [ -f /tmp/my_tmp_file_$$ ]; do
  echo File exists
  sleep 1
done
echo The file no longer exists

trap - INT
echo creating file /tmp/my_tmp_file_$$
date > /tmp/my_tmp_file_$$

echo "press interrupt (Ctrl-C) to interrupt ...."
while [ -f /tmp/my_tmp_file_$$ ]; do
  echo File exists
  sleep 1
done

echo we never get here
exit 0
```

スクリプトを実行し、各ループの実行中に**Ctrl - C**（または割り込みを発生させるキー）を押すと、次のように出力されます。

```
creating file /tmp/my_tmp_file_141
press interrupt (Ctrl-C) to interrupt ....
File exists
File exists
File exists
File exists
The file no longer exists
creating file /tmp/my_tmp_file_141
press interrupt (Ctrl-C) to interrupt ....
File exists
File exists
File exists
File exists
File exists
```

解説

このスクリプトでは、最初に**trap**コマンドを使って、**INT**（割り込み）シグナルの発生時に**rm -f /tmp/my_tmp_file_$$**というコマンドを実行するようにしています。次に、**/tmp_my_tmp_file_$$**というファイルが存在する間、繰り返し実行される**while**ループがあります。ユーザーが**Ctrl - C**キーを押すと、**rm -f /tmp/my_tmp_file_$$**が実行され、**while**ループが再開されます。しかし、ファイルは削除されているので、再開後の最初の条件判定で**while**ループが終了します。

このあと、再び**trap**コマンドを使用し、今度は**INT**シグナルが発生してもなにもコマンドを実行しないようにしています。この設定のあと、ファイルを再作成して2つ目の**while**ループに入ります。ユーザーが**Ctrl - C**キーを押しても、今度は実行すべきコマンドが設定されていないので、デフォルトの動作が実行され、スクリプトはただちに終了します。ユーザーが**Ctrl - C**キーを押した時点で即座にスクリプトが終了するので、最後の**echo**ステートメントと**exit**ステートメントは実行されません。

◆ unset

unsetコマンドは、変数または関数を環境から削除します。ただし、環境変数などの読み取り専用変数は削除できません。このコマンドはあまり使われません。

```
#!/bin/sh

foo="Hello World"
echo $foo

unset foo
echo $foo
```

このスクリプトを実行すると、**Hello World**が初めの1回だけ出力され、次は改行文字だけが出力されます。

> **メモ**
> 上のスクリプトでは、**foo=**と記述しても**unset**を使った場合と同じ結果が得られますが、**foo**に空文字列を設定することと、環境から**foo**を削除することは同じではありません。

2.4.6 コマンドの実行

スクリプトを記述する際、コマンドの実行結果を取り込まなければならないことがあります。たとえば、シェルスクリプトでコマンドを実行し、コマンドからの出力を変数に設定したい場合などです。このような場合には、**set**コマンドの例で使った**$(command)**という構文を利用します。**`command`**は同じ構文の古い形式で、こちらも依然よく使われます。

> **注意**
> コマンドを実行するための古い形式の構文では、単一引用符(')ではなく、バッククォート(`)を使うことに注意してください。単一引用符は、変数をクォートし、変数が値に展開されるのを防ぐために使います。移植性を高くしたい場合には、古い形式の構文を使うとよいでしょう。

$(...)**という新しい形式が導入された背景には、バッククォートで囲まれたコマンド内での**$、**`**、**\\**の扱いがわかりにくいという事情があります。たとえば、**`...`**の中でバッククォートを使う場合には、バッククォートを****でエスケープしなければなりません。このため、経験を積んだシェルプログラマでも、バッククォートで囲まれたコマンドで正しくクォートを行うのに苦労することがあります。

$(command)の結果は、コマンドからの出力になります。これは、コマンドの終了ステータスではなく、コマンドからの文字列出力です。たとえば、次の例を見てください。

```
#!/bin/sh

echo The current directory is $PWD
echo The current users are $(who)

exit 0
```

現在のディレクトリはシェルの環境変数なので、最初の行ではコマンド実行の構文を使う必要はありません。しかし、**who**の実行結果を取得してスクリプトで利用するには、コマンド実行の構文を使う必要があります。

コマンドの実行結果をスクリプトの変数に割り当てる機能は非常に強力です。この機能を使えば、スクリプトからコマンドを実行し、その出力を簡単に取り込むことができます。

ただし、実行したコマンドからの出力を受け取る際、目的の文字列の前にホワイトスペースが含まれていたり、実際に必要なもの以上の出力が返される場合には、なんらかの処理が必要になることがあります。このような場合には、すでに説明した**set**コマンドを使うと、出力の中から目的の文字列を取り出すことができます。

◆ 算術展開

すでに取り上げた**expr**コマンドを使えば、単純な算術演算は可能です。しかし、**expr**コマンドを処理するために新しいシェルを呼び出す必要があり、効率はあまりよくありません。

このような場合には、代わりに**$((...))**による展開を使うと便利です。評価したい式を**$((...))**で囲むだけで、単純な算術演算を効率よく実行することができます。

```
#!/bin/sh

x=0
while [ "$x" -ne 10 ]; do
  echo $x
  x=$(($x+1))
done

exit 0
```

◆ パラメータ展開

パラメータの割り当てと展開については、次のようなごく単純なケースを取り上げてすでに説明しました。

```
foo=fred
echo $foo
```

では、変数のあとになんらかの文字列を追加したい場合にはどうすればよいのでしょうか。たとえば、**1_tmp**、**2_tmp**のような名前の付いた複数のファイルをスクリプトで処理したい場合などです。まず試しに、次のようなスクリプトを書いてみます。

```
#!/bin/sh

for i in 1 2
do
    my_secret_process $i_tmp
done
```

メモ

もちろん、**my_secret_process**というコマンドで複数の引数を処理できるようにしておくことも可能ですが、ここではパラメータの展開について説明することを目的としているので、とりあえず**my_secret_process**は引数を1つだけ取るコマンドと考えてください。

このスクリプトを実行すると、次のようなメッセージが出力されます。

```
my_secrt_process: too few arguments
```

どこがいけないのでしょうか。問題は、スクリプトの中で**1_tmp**や**2_tmp**に置き換えられることを期待している変数**$i_tmp**が、実際には存在していないことです。この変数の**$i**の部分が展開されるようにするには、**i**を**{}**で囲んで次のように記述する必要があります。

```
#!/bin/sh

for i in 1 2
do
    my_secret_process ${i}_tmp
done
```

今度は、ループを回るたびに**${i}**が**i**の値に置き換えられ、目的のファイル名が得られます。この方法を使えば、パラメータの値を文字列の中に埋め込むことができます。

シェルでは、さまざまな**パラメータ置換**を行うことができます。パラメータ置換を使うと、パラメータ処理に関するさまざまな問題をエレガントに解決することができます。

次の表は、よく使うパラメータ置換についてまとめたものです。

表2.12 パラメータ置換

パラメータ展開	説明
`${param:-default}`	**param**が空ならば、**default**の値を返す。
`${#param}`	**param**の長さを返す。
`${param%word}`	**param**の末尾を起点として、**word**に一致する**param**の最短部分を削除し、残りの部分を返す。
`${param%%word}`	**param**の末尾を起点として、**word**に一致する**param**の最長部分を削除し、残りの部分を返す。
`${param#word}`	**param**の先頭を起点として、**word**に一致する**param**の最短部分を削除し、残りの部分を返す。
`${param##word}`	**param**の先頭を起点として、**word**に一致する**param**の最長部分を削除し、残りの部分を返す。

これらの置換は、文字列を操作するときに役立ちます。特に最後の4つは、ファイル名やパスを扱うときに便利です。次に実際の例を示します。

例題 パラメータ処理

次のスクリプトは、さまざまなパラメータ一致演算子を使った例です。

```
#!/bin/sh

unset foo
echo ${foo:-bar}

foo=fud
echo ${foo:-bar}

foo=/usr/bin/X11/startx
echo ${foo#*/}
echo ${foo##*/}

bar=/usr/local/etc/local/networks
echo ${bar%local*}
echo ${bar%%local*}

exit 0
```

このスクリプトの出力は次のようになります。

```
bar
fud
usr/bin/X11/startx
startx
/usr/local/etc/
/usr/
```

解説

最初のステートメントが実行された時点では、fooは値を持っていないので、${foo:-bar}は値barになります。fooという変数は設定されないままなので、fooの値は変わりません。

メモ

スクリプトの中で取り上げたもののほかに、次のようにさまざまなパラメータ展開を利用することができます。

${foo:-bar}ではなく、${foo:=bar}を使うと、変数に$fooが設定されます。この文字列演算子は、fooが存在し、しかも空ではないことを確認します。実際にfooが存在していて空でない場合には、fooの値を返します。そうでない場合には、fooにbarを設定し、その値を返します。

${foo:?bar}は、fooが存在しないか、または空の場合にはfoo: barと出力し、コマンドをアボートします。

${foo:+bar}は、fooが存在していて空でない場合には、barを返します。

{foo#*/}では、一番左の/がパターンに一致し、これだけが削除されます（*は0個以上の文字に一致します）。{foo##*/}では、パターンに一致する最長部分が削除されます。このため、一番右の/までのすべての文字が削除されることになります。

{bar%local*}では、右を起点にlocal（と、それに続く任意の個数の文字）に初めて一致する部分までが削除されます。{bar%%local*}では、右を起点にlocalに一致する最長部分、つまり一番左のlocalに一致する部分までが削除されます。

UNIXではフィルタを使うことが多いので、しばしば操作結果のリダイレクトが必要になります。たとえば、Slackwareに含まれているcjpegというフィルタを使ってgifファイルをjpegファイルに変換したい場合には、次のようなコマンドを実行します。

```
$ cjpeg image.gif > image.jpg
```

しかし、数多くのファイルに対してこの種の処理を実行したいこともあります。このような場合には、次のようにパラメータ展開を使うと、非常に簡単に目的の処理を実行することができます。

```
#!/bin/sh
for image in *.gif
do
  cjpeg $image > ${image%%gif}jpg
done
```

このスクリプトをgiftojepgという名前で保存すれば、現在のディレクトリの各gifファイルに対してjpegファイルを作成することができます。

2.4.7 ヒヤドキュメント

コマンドへの入力をシェルスクリプトから渡す特別な方法として、**ヒヤドキュメント**（here document）があります。ヒヤドキュメントを使うと、スクリプトの中で実行するコマンドに対して、ファイルやキーボードから入力を与える場合と同様に、スクリプトから入力を渡すことができます。

ヒヤドキュメントは、リーダー<<で始まり、そのあとになんらかの特別な文字列が続きます。この特別な文字列は、ドキュメントの最後でもう一度繰り返されます。<<は、シェルのラベルリダイレクタで、ヒヤドキュメントの場合には、コマンドへの入力をヒヤドキュメントによって行うことを意味します。<<のあとの文字列は、ヒヤドキュメントの終了位置をシェルに指示するためのマーカーとして機能します。マーカー文字列はコマンドに渡す行の中では使用できないので、通常は、ほかの文字列とすぐに見分けがつく特別な文字列を使用します。

例題　ヒヤドキュメントの使用

最も簡単な例として、catコマンドに入力を渡します。

```
#!/bin/sh
cat <<!FUNKY!
hello
this is a here
document
!FUNKY!
```

出力は次のようになります。

```
hello
this is a here
document
```

ヒヤドキュメントはかなり特異な印象を与えるかもしれませんが、実際には応用範囲の広い強力な機能です。たとえば、エディタなどの対話的なプログラムを呼び出し、あらかじめ指定した入力を与えることができます。ただし、一般的には、上の例のようにスクリプトの中で大量のテキストを出力するのに使います。ヒヤドキュメントを使えば、各行でechoコマンドを使わなくてもテキストを出力できるからです。上の例では、ヒヤドキュメントの識別子の両側に!を付けて、ほかの文字列と容易に区別できるようにしています。

例題　ヒヤドキュメントのその他の使い方

1 まず、次のような内容のa_text_fileを用意します。

```
That is line 1
That is line 2
That is line 3
That is line 4
```

2 ヒヤドキュメントとedエディタを組み合わせてこのファイルを編集します。次のようなスクリプトを記述します。

```
#!/bin/sh

ed a_text_file<<!FunkyStuff!
3
d
.,\$s/is/was/
w
q
!FunkyStuff!
```

スクリプトを実行したあと、`a_text_file`の内容は次のようになります。

```
That is line 1
That is line 2
That was line 4
```

解説

スクリプトから`ed`エディタを起動し、`ed`エディタへのコマンドを渡します。コマンドの内容は、`a_text_file`の3行目に移動してその行を削除し、次に現在の行（3行目は削除されているのでもとのファイルの4行目）の`is`を`was`に置き換えるというものです。このコマンドは、スクリプト内のヒヤドキュメント、つまりマーカー`!FunkyStuff!`で囲まれた部分の行から渡されます。

> **注意**
> ヒヤドキュメントの内部では、`\`を使って、`$`がシェルによって展開されないように保護しています。`\`は`$`をエスケープするので、シェルは`$s/is/was/`を値に展開することはしません。`\$`は`$`として`ed`エディタに渡され、`ed`エディタが`$`を解釈します。

2.4.8 スクリプトのデバッグ

シェルスクリプトのデバッグはそれほど難しくありませんが、デバッグ用の特別なツールのようなものは存在しません。以下では、よく使うデバッグのテクニックについて簡単に説明します。

通常、シェルは、エラーが発生すると、エラーを含む行の行番号を表示します。エラーの原因がすぐにわからない場合には、`echo`ステートメントをいくつか追加して変数の内容を表示したり、疑わしいコードの一部を対話的にシェルに入力して動作をテストすることができます。

スクリプトはシェルによって解釈されるだけなので、スクリプトを修正して再実行する場合でもコンパイルのためのオーバーヘッドはありません。

シェルにはさまざまなオプションが用意されており、これらのオプションを使うと、原因の特定が困難なエラーをトレースすることができます。シェルのオプションを指定するには、シェルを呼び出したあとでコマンドラインオプションを使う方法と、`set`コマンドを使う方法があります。次の表は、シェルのオプションをまとめたものです。

表2.13 シェルのオプション

コマンドラインオプション	set オプション	説明
sh -n <script>	set -o noexec set -n	構文エラーだけをチェックし、コマンドは実行しない。
sh -v <script>	set -o verbose set -v	実行する前にコマンドをエコーする。
sh -x <script>	set -o xtrace set -x	コマンドラインで処理したあとにコマンドをエコーする。
-	set -o nounset set -u	未定義の変数が使われている場合にエラーメッセージを表示する。

set オプションのフラグをオンにするには **-o** を、オフにするには **+o** を使用します（**set -n**、**set -v** などの短縮形の場合にも、同様の指定でオンとオフを切り替えます）。

xtrace オプションを使うと、簡単な実行トレースを行うことができます。最初はコマンドラインオプションを使ってスクリプトをチェックするとよいでしょう。問題のある箇所の見当がついたら、その前後に **xtrace** フラグを設定するコマンドを入力します（問題の箇所の前でオンにして、あとでオフにします）。実行トレースを行うと、実際に実行される行が、変数を展開した状態で（その行を実行する前に）表示されます。デフォルトでは、展開のレベルは、各行の先頭に表示されるプラス記号（**+**）の数によって表されます。シェルの設定ファイルでシェル変数 **PS4** を設定すれば、**+** を任意のものに変えることができます。

シェルでは、**EXIT** シグナルをトラップすることで、終了時のプログラムの状態を調べることができます。それには、スクリプトの始めのほうに次のような行を記述しておきます。

```
trap 'echo Exiting: critical variable = $critical_variable' EXIT
```

2.5 実用的なプログラム

この章では、プログラミング言語としてのシェルの主要機能について説明しました。ここではそのおさらいとして、シェルの機能を使ってサンプルプログラムを作成します。

本書では、これ以後、各章で学んだテクニックを応用してCDデータベースアプリケーションを作成していきます。最初はシェルスクリプトでアプリケーションを記述しますが、以後は各章の内容に応じてC言語で書き直したり、データベースを追加するなど、アプリケーションに修正を加えていきます。

2.5.1 仕様

CDを管理するためのプログラムを設計し、実装します。大量のCDコレクションを管理するには電子的なカタログが不可欠ですが、これをプログラムとして実装する試みは、UNIXプログラミングを学習するうえで貴重な経験となるはずです。

まず、CDのラベル、ジャンル、アーティストや作曲家など、各CDについての基本的な情報を格納する必要があります。また、CDの各トラック（曲）に関する情報も扱えるようにすると便利でしょう。

検索はCD単位で行えるようにします。トラックについては、とりあえず今は考慮しません。

アプリケーションとして使いやすいものにするには、アプリケーションの内部ですべての情報の入力、更新、削除を行えるようにする必要があります。

2.5.2 設計

データの更新、検索、表示という3つの仕様を実現するには、簡単なメニューを用意するとよいでしょう。保存する必要のあるデータはすべてテキスト形式とします。コレクションに含まれるCDの数がそれほど膨大ではないと仮定すれば、単純なテキストファイルで十分目的は達成できるので、複雑なデータベースは不要です。テキストファイルに情報を格納すれば、アプリケーションもシンプルになります。また、仕様を変更した場合でも、単純なテキストファイルのほうが独自形式のファイルより修正は容易です。場合によっては、プログラムを書かなくても、エディタを使ってデータを手作業で入力したり削除したりすることができます。

データの格納方法については、設計上重要な決定を下す必要があります。それは、1つのファイルにデータを格納すべきか、また、そうだとすればどのような形式で格納すべきかということです。トラック情報を除けば、格納する必要のあるほとんどの情報は1枚のCDにつき1つです（CDによっては複数の作曲家やアーティストの曲を含んでいることがありますが、とりあえずここでは考えないことにします）。また、通常、1枚のCDには複数のトラックがあります。

1枚のCDに関して格納できるトラック情報の数は制限すべきでしょうか。情報の一部を意図的に欠落させるこうした制限は、特に必要なわけでもありませんし、現実的でもないので、ここでは制限を設けないことにします。

任意の数のトラック情報を格納できるとした場合、ファイルの形式については次の3つの選択肢が考えられます。

- 単一のファイルを使用する。タイトルなどの情報は1行に記述し、トラック情報は複数の行に分けて記述する。
- 各CDの情報をすべて1行で記述する。トラック情報もこの行の中に含めて記述する。
- タイトル情報とトラック情報を分け、それぞれ別のファイルに記述する。

上の3つの選択肢のうち、ファイルの形式が常に一定となるのは3つ目の選択肢です。ファイルの形式が不変ならば、データベースをあとでリレーショナルな形式に変換するときにも余計な手間をかけずにすみます（リレーショナルな形式への変換については、実際に第7章で取り上げます）。そこで、ここでは3番目の選択肢を採用することにします。

次に、ファイルに格納する情報を決める必要があります。

各CDタイトルについては、次の情報を格納することにします。

- CDカタログ番号
- タイトル
- ジャンル（クラシック、ロック、ポップス、ジャズなど）
- 作曲家またはアーティスト

トラックについては、次の情報を格納します。

- トラック番号
- 曲名

CDタイトルのファイルとトラック情報のファイルを結び付けるには、トラック情報をCDの情報と関連付ける必要があります。これにはカタログ番号を使います。カタログ番号は各CDに固有の情報なので、タイトルファイル内で重複することはありません。また、トラックファイルについても、各トラックに対応するカタログ番号は1つだけです。

タイトルファイルは次のような内容で作成します。

表2.14　タイトルファイル（CD情報）

カタログ	タイトル	ジャンル	作曲家
CD123	Cool sax	Jazz	Bix
CD234	Classic violin	Classical	Bach
CD345	Hits91	Pop	Various

トラックファイルの内容は次のとおりです。

表2.15　トラックファイル（曲情報）

カタログ	トラック番号	タイトル
CD123	1	Some jazz
CD123	2	More jazz
CD345	1	Dizzy
CD234	1	Sonata in D minor

これら2つのファイルは、“カタログ”フィールドを使って結合します。トラックファイルには、タイトルファイル内の1つのエントリに対して複数の行が存在する可能性があることに注意する必要があります。

さて、最後に、各エントリの区切りをどうするかについても決める必要があります。リレーショナルデータベースでは固定長フィールドがよく使われますが、固定長フィールドは必ずしも扱いやすいわけではありません。ここでは、別の方法としてカンマで項目を区切り、CSV形式のファイルを使うことにします（CSVは、comma-separated valueの略です）。

以下では、実際のスクリプトの内容を見ていきます。少し長いので、全体を見失うことがないように、スクリプトの中で使われている関数の一覧を最初にあげておきます。

```
get_return()
get_confirm()
set_menu_choice()
insert_title()
insert_track()
add_record_tracks()
add_records()
find_cd()
update_cd()
count_cds()
remove_records()
list_tracks()
```

例題 CDアプリケーション

1 シェルスクリプトとして実行するのに必要な設定をスクリプトの1行目に記述し、そのあとに著作権情報を記述します。

```
#!/bin/sh

# Very simple example shell script for managing a CD collection.
# Copyright (C) 1996 Wrox Press.

# This program is free software; you can redistribute it and/or modify it
# under the terms of the GNU General Public License as published by the
# Free Software Foundation; either version 2 of the License, or (at your
# option) any later version.

# This program is distributed in the hopes that it will be useful, but
# WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General
# Public License for more details.

# You should have received a copy of the GNU General Public License along
# with this program; if not, write to the Free Software Foundation, Inc.
# 675 Mass Ave, Cambridge, MA 02139, USA.
```

2 次に、スクリプト全体を通じて使用するグローバル変数をセットアップします。具体的には、タイトルファイルとトラックファイル、それに作業用の一時ファイルを設定します。また、**Ctrl - C**キーをトラップし、ユーザーによってスクリプトの実行が中断されたときに一時ファイルを削除できるようにします。

```
menu_choice=""
current_cd=""
title_file="title.cdb"
tracks_file="tracks.cdb"
temp_file=/tmp/cdb.$$
trap 'rm -f $temp_file' EXIT
```

3 必要な関数をすべて定義します。スクリプトは1行目から実行されるので、スクリプトの先頭ですべての関数を定義しておけば、どの関数も呼び出す前に必ず定義されていることになります。

最初の2つの関数は、同じコードを繰り返し記述するのを避けるために用意する簡単なユーティリティ関数です。

```
get_return() {
  echo -e "Press return \c"
  read x
  return 0
}

get_confirm() {
  echo -e "Are you sure? \c"
  while true
  do
    read x
    case "$x" in
      y | yes | Y | Yes | YES )
        return 0;;
      n | no  | N | No  | NO )
        echo
        echo "Cancelled"
        return 1;;
      *) echo "Please enter yes or no" ;;
    esac
  done
}
```

4 次は、メインメニュー関数`set_menu_choice`です。メニューの内容は動的に変化し、CDエントリが選択されている場合には追加のオプションが表示されます。この関数で使われている`echo -e`は、シェルによっては使用できない可能性もあります。

```
set_menu_choice() {
  clear
  echo "Options :-"
  echo
  echo " a) Add new CD"
  echo " f) Find CD"
  echo " c) Count the CDs and tracks in the catalog"
  if [ "$cdcatnum" != "" ]; then
    echo "   l) List tracks on $cdtitle"
    echo "   r) Remove $cdtitle"
    echo "   u) Update track information for $cdtitle"
  fi
  echo " q) Quit"
  echo
  echo -e "Please enter choice then press return \c"
  read menu_choice
  return
}
```

5 2つの短い関数insert_titleとinert_trackは、データベースファイルにデータを追加します。たかだか1行で記述できる内容を関数にする意味があるのかと疑問を感じるかもしれませんが、こうしておくことで、ほかの関数をわかりやすく記述することができます。

insert_titleとinsert_trackは、次の少し長めのadd_record_track関数で使われています。add_record_track関数では、パターンマッチを使って入力内容にカンマが含まれないようにします（カンマはフィールドセパレータとして使うので、データにカンマを含めることはできません）。また、トラック情報が入力されるごとに、算術演算を使って現在のトラック番号を1つ増やします。

```
insert_title() {
  echo $* >> $title_file
  return
}

insert_track() {
  echo $* >> $tracks_file
  return
}

add_record_tracks() {
  echo "Enter track information for this CD"
  echo "When no more tracks enter q"
  cdtrack=1
  cdttitle=""
  while [ "$cdttitle" != "q" ]
  do
      echo -e "Track $cdtrack, track title? \c"
      read tmp
      cdttitle=${tmp%%,*}
      if [ "$tmp" != "$cdttitle" ]; then
```

```
        echo "Sorry, no commas allowed"
        continue
      fi
      if [ -n "$cdttitle" ]; then
        if [ "$cdttitle" != "q" ]; then
          insert_track $cdcatnum,$cdtrack,$cdttitle
        fi
      else
        cdtrack=$((cdtrack-1))
      fi
    cdtrack=$((cdtrack+1))
  done
}
```

6 add_records関数は、新しいCD情報のエントリを作成するために使います。

```
add_records() {
  # Prompt for the initial information

  echo -e "Enter catalog number \c"
  read tmp
  cdcatnum=${tmp%%,*}

  echo -e "Enter title \c"
  read tmp
  cdtitle=${tmp%%,*}

  echo -e "Enter type \c"
  read tmp
  cdtype=${tmp%%,*}

  echo -e "Enter artist/composer \c"
  read tmp
  cdac=${tmp%%,*}

  # Check that they want to enter the information

  echo About to add new entry
  echo "$cdcatnum $cdtitle $cdtype $cdac"

  # If confirmed then append it to the titles file

  if get_confirm ; then
    insert_title $cdcatnum,$cdtitle,$cdtype,$cdac
    add_record_tracks
  else
    remove_records
  fi

  return
}
```

7 find_cd関数は、grepコマンドを使ってCDタイトルファイルから指定された文字列を検索します。このとき、検索文字列に一致する行がいくつあるかを知る必要がありますが、grepコマンドが返す値からわかるのは、一致する文字列があったかどうかだけです。そこで、grepコマンドの出力をいったんファイルに格納し、このファイルの行数を数えます。

ワードカウントコマンドwcからの出力にはホワイトスペースが含まれ、これが、ファイルに含まれる行数、ワード数、文字数の区切りとして使われます。このため、まず$(wc -l $temp_file)によってファイルの行数をカウントしたあと、変数linesfoundにファイルの行数を設定します。setは、コマンドからの出力をシェルのパラメータ変数に設定するために使います。

次に、IFS（内部フィールドセパレータ）をカンマ（,）に変更し、カンマ区切りのフィールドを分離できるようにします。IFSを変更する代わりにcutコマンドを使う方法もあります。

```
find_cd() {
  if [ "$1" = "n" ]; then
    asklist=n
  else
    asklist=y
  fi
  cdcatnum=""
  echo -e "Enter a string to search for in the CD titles \c"
  read searchstr
  if [ "$searchstr" = "" ]; then
    return 0
  fi

  grep "$searchstr" $title_file > $temp_file

  set $(wc -l $temp_file)
  linesfound=$1

  case "$linesfound" in
  0)    echo "Sorry, nothing found"
        get_return
        return 0
        ;;
  1)    ;;
  2)    echo "Sorry, not unique."
        echo "Found the following"
        cat $temp_file
        get_return
        return 0
  esac

  IFS=","
  read cdcatnum cdtitle cdtype cdac < $temp_file
  IFS=" "

  if [ -z "$cdcatnum" ]; then
```

```
    echo "Sorry, could not extract catalog field from $temp_file"
    get_return
    return 0
  fi

  echo

  echo Catalog number $cdcatnum
  echo $cdtitle
  echo $cdtype
  echo $cdac
  echo
  get_return

  if [ "$asklist" = "y" ]; then
    echo -e "View tracks for this CD? \c"
      read x
    if [ "$x" = "y" ]; then
      echo
      list_tracks
      echo
    fi
  fi
  return 1
}
```

8 update_cdは、CD情報を再入力するための関数です。この関数では、get_confirmがtrueを返した場合に実行する複数のコマンドを、{}を使ってグループ化しています。

```
update_cd() {
  if [ -z "$cdcatnum" ]; then
    echo "You must select a CD first"
    find_cd n
  fi
  if [ -n "$cdcatnum" ]; then
    echo "Current tracks are :-"
    list_tracks
    echo
    echo "This will re-enter the tracks for $cdtitle"
    get_confirm && {
      grep -v "^$cdcatnum" $tracks_file > $temp_file
      mv $temp_file $tracks_file
#       cat $temp_file > $tracks_file
      echo
      add_record_tracks
    }
  fi
  return
}
```

9 count_cdsは、データベースに格納されている情報をカウントします。

```
count_cds() {
  set $(wc -l $title_file)
  num_titles=$1
  set $(wc -l $tracks_file)
  num_tracks=$1
  echo found $num_titles CDs, with a total of $num_tracks tracks
  get_return
  return
}
```

10 remove_recordsは、データベースファイルからエントリを削除します。具体的には、grep -vを使って、検索文字列を含む行を取り除きます。このとき一時ファイルを使う必要があることに注意してください。
たとえば、次のように実行した場合には問題が起きます。

```
grep -v "^$cdcatnum" $title_file > $title_file
```

このコマンドでは、grepを実行する前に、>による出力のリダイレクトで$title_fileが空になるので、空のファイルに対してgrepコマンドを実行することになってしまいます。

```
remove_records() {
  if [ -z "$cdcatnum" ]; then
    echo You must select a CD first
    find_cd n
  fi
  if [ -n "$cdcatnum" ]; then
    echo "You are about to delete $cdtitle"
    get_confirm && {
      grep -v "^$cdcatnum" $title_file > $temp_file
      mv $temp_file $title_file
      grep -v "^$cdcatnum" $tracks_file > $temp_file
      mv $temp_file $tracks_file
      cdcatnum=""
      echo Entry removed
    }
    get_return
  fi
  return
}
```

11 list_tracksでは、grepコマンドを使って目的の行を取り出し、cutコマンドで必要なフィールドを切り出したあと、moreコマンドで1画面ごとに結果を出力します。わずか20行程度の関数ですが、同じ処理をCで記述するとなると、かなりの行数が必要になるでしょう。この点からも、シェルが非常に強力なツールであることがわかります。

```
list_tracks() {
  if [ "$cdcatnum" = "" ]; then
    echo no CD selected yet
    return
  else
    grep "$cdcatnum" $tracks_file > $temp_file
    set $(wc -l $temp_file)
    num_tracks=$1
    if [ "$num_tracks" = "0" ]; then
      echo no tracks found for $cdtitle
    else {
      echo
      echo "$cdtitle :-"
      echo
      cut -f 2- -d , $temp_file
      echo }| more
    fi
  fi
  get_return
  return
}
```

12 これですべての関数の定義を終えました。次はメインルーチンです。まず冒頭でファイルが確実に存在するようにしてから、メニュー関数`set_menu_choice`を呼び出し、ユーザーからの入力に応じて必要な処理を実行します。
終了が選択された場合には、一時ファイルを削除し、メッセージを表示して終了コード0でプログラムを終了します。

```
rm -f $temp_file

if [ ! -f $title_file ]; then
  touch $title_file
fi
if [ ! -f $tracks_file ]; then
  touch $tracks_file
fi

# Now the application proper

clear
echo
echo
echo "Mini CD manager"
sleep 3

quit=n
while [ "$quit" != "y" ];
do
  set_menu_choice
  case "$menu_choice" in
```

```
      a) add_records;;
      r) remove_records;;
      f) find_cd y;;
      u) update_cd;;
      c) count_cds;;
      l) list_tracks;;
      b)
        echo
        more $title_file
        echo
        get_return;;
      q | Q ) quit=y;;
      *) echo "Sorry, choice not recognized";;
    esac
done

#Tidy up and leave

rm -f $temp_file
echo "Finished"
exit 0
```

解説

スクリプトの最初に記述したtrapコマンドは、ユーザーが押したCtrl-Cキーをトラップするためのものです。生成されるシグナルは、端末の設定によってはEXITではなくINTになることがあります。

メニュー選択を実装するには、ほかにもさまざまな方法があります。たとえば、bashやkshで利用できるメニュー選択構文のselectを使う方法もあります（ただし、これはX/Open仕様にはない機能です）。移植性が多少低下してもかまわない場合には、これらの方法も検討するとよいでしょう。また、複数の行にまたがる情報をユーザーに提示するときには、ヒヤドキュメントを利用することもできます。

このスクリプトでは、新しいレコードを入力するときに主キーのチェックを行っていません。このため、すでに存在するカタログ番号を使ってCDタイトルを入力した場合、そのCDに含まれるトラックは、次のように既存のタイトルのトラックとして表示されることになります。

```
1 First CD Track 1
2 First CD Track 2
1 Another CD
2 With the same CD key
```

ここでは、これらの問題に対する解決方法は示しません。このサンプルプログラムはGPLのもとで自由に改変できるので、興味があったら自分でスクリプトを改善してみてください。

2.6 この章のまとめ

この章では、強力なプログラム言語としてのシェルの使い方を学びました。ほかのプログラムを手軽に呼び出して出力を処理できるシェルは、テキスト処理やファイル操作を行うのに理想的なツールといえます。

小さなユーティリティプログラムが必要になった場合には、さまざまなUNIXのコマンドを組み合わせたシェルスクリプトで実現可能かどうか考えてみるとよいでしょう。シェルを使えば、コンパイラの助けを借りなくても、多くの便利なユーティリティを作成することができます。



第3章

ファイルの操作

この章では、UNIXのファイルとディレクトリについて説明し、ファイルの作成、オープン、読み取り、書き込み、クローズの方法を学びます。また、ディレクトリの作成、走査、削除など、プログラムからディレクトリを操作する方法についても学習します。第2章では少しまわり道をしてシェルについて詳しく説明しましたが、この章からCによるプログラミングの学習を開始します。

まず最初に、ファイル、ディレクトリ、デバイスといった概念についてひととおり解説します。次に、UNIXでのファイル入出力の扱いについて詳しく見ていきます。ファイルとディレクトリを操作するには、システムコール（WinodwsのAPIに相当）を実行する必要がありますが、こうしたシステムコールのほかにも、ファイル操作をより効率的に行うためのライブラリ関数が数多く用意されています。

この章で取り上げる項目は次のとおりです。

- ファイルとデバイス
- システムコール
- ライブラリ関数
- 低水準ファイルアクセス
- ファイルの管理
- 標準入出力ライブラリ
- 書式付き入出力
- ファイルとディレクトリの保守
- ディレクトリの走査
- エラー
- その他のファイル操作

3.1 UNIXのファイル構造

すでによく知られているファイル構造について、なぜあらためて取り上げる必要があるのか不思議に思われるかもしれません。しかし、ファイルは、オペレーティングシステムサービスとデバイスに対するシンプルで一貫したインタフェースを提供するものであり、UNIX環境では特に重要な意味を持っています。それは一言でいえば、**“すべてがファイルである”**ということです。

すべてがファイルであるということは、ディスクファイルやシリアルポート、プリンタ、その他のデバイスを、ファイルを扱うのと同様にプログラムから扱うことができるということです。いくつかの例外についてはあとでまた取り上げますが、通常必要になるのは`open`、`close`、`read`、`write`、`ioctl`という5つの基本関数だけです。

ディレクトリも特殊な種類のファイルです。UNIXの初期バージョンでは、スーパーユーザーでもディレクトリへの書き込みはできず（読み取りは可能）、ディレクトリを削除するには、ディレクトリファイルを空にして`rmdir`を使う必要がありました。しかし、新しい高水準の`opendir`／`readdir`インタフェースでは、より洗練された形でディレクトリを操作できるようになっています。ディレクトリ関連の特別な関数については、あとで詳しく取り上げます。

さて、UNIXでは実際にすべてのものがファイルとして表現されます。もちろん、ファイルが表しているものによっては、ふだん使っているファイルと微妙に異なる部分もありますが、すべてのものがファイルであるという大原則は常に有効です。次に、特別なケースについて少し説明しておきましょう。

3.1.1 ディレクトリ

ファイルは、その内容のほかに、名前と管理情報（ファイルの作成／修正日時とパーミッション）を持っています。管理情報が格納されているのは**iノード**で、このiノードには、ファイルの長さとディスク上でのファイルの位置に関する情報も含まれます。システムが使用するのはファイルのiノードの番号です。ディレクトリ構造は、ユーザーの便宜のためにファイルの名前を提供する役割を果たしています。

ディレクトリとは、ほかのファイルのiノードと名前を保持しているファイルのことです。UNIXのバージョンによっては、8進数ダンプユーティリティの`od`コマンドを使って、`od -c .`でファイル名を、`od -d .`でiノードを表示できるものがあります。各ディレクトリエントリは、ファイルのiノードへのリンクです。ファイル名を削除すると、このリンクが削除されます。`ln`コマンドを使うと、同じファイルへのリンクをさまざまなディレクトリに作成することができます。ファイルへのリンクの数（`ls -l`を実行したときにパーミッションの次に表示される数字）が0になると、該当するiノードとデータは使われていないことになり、空きとしてマークされます。

ファイルはディレクトリに置かれますが、ディレクトリにはさらにサブディレクトリが含まれていることがあります。このしくみによって、いわゆる**ファイルシステムの階層構造**が形作られます。たとえば、`neil`というユーザーがいたとします。ふつう、`neil`は自分のファイルをホームディレクトリに置きます。ホームディレクトリは、一般的には`/home/neil`になるでしょう。ホームディレクトリには、メール用、ビジネスレター用、ユーティリティプログラム用などのサブディレクトリを置くことができます。UNIXには便利な表記方法があり、チルダ（~）を使うと、自分のホームディレクトリに簡単に移動することができます（たとえば、ユーザー`user`の場合には、`~user`と入力します）。一般に、各ユーザーのホームディレクトリは、それより上位のディレクトリのサブディレクトリになっています。上位のディレクトリは、ユーザーのホームディレクトリを格納するために特別に作成し、`/home`などの名前を付けるのが一般的です。

次の図で、`/home`ディレクトリは、それ自体がルートディレクトリ（`/`）のサブディレクトリに

なっています。**ルートディレクトリ**は階層構造の頂点に位置し、システムのすべてのファイルはそのサブディレクトリに収められています。通常、ルートディレクトリには、システムプログラムを置くための**/bin**（バイナリの意）、システム設定ファイルを置いておく**/etc**、システムライブラリを置くための**/lib**などのサブディレクトリがあります。物理デバイスを表すファイルや、デバイスへのインタフェースを提供するファイルは、伝統的に**/dev**という名前のディレクトリに置かれます。

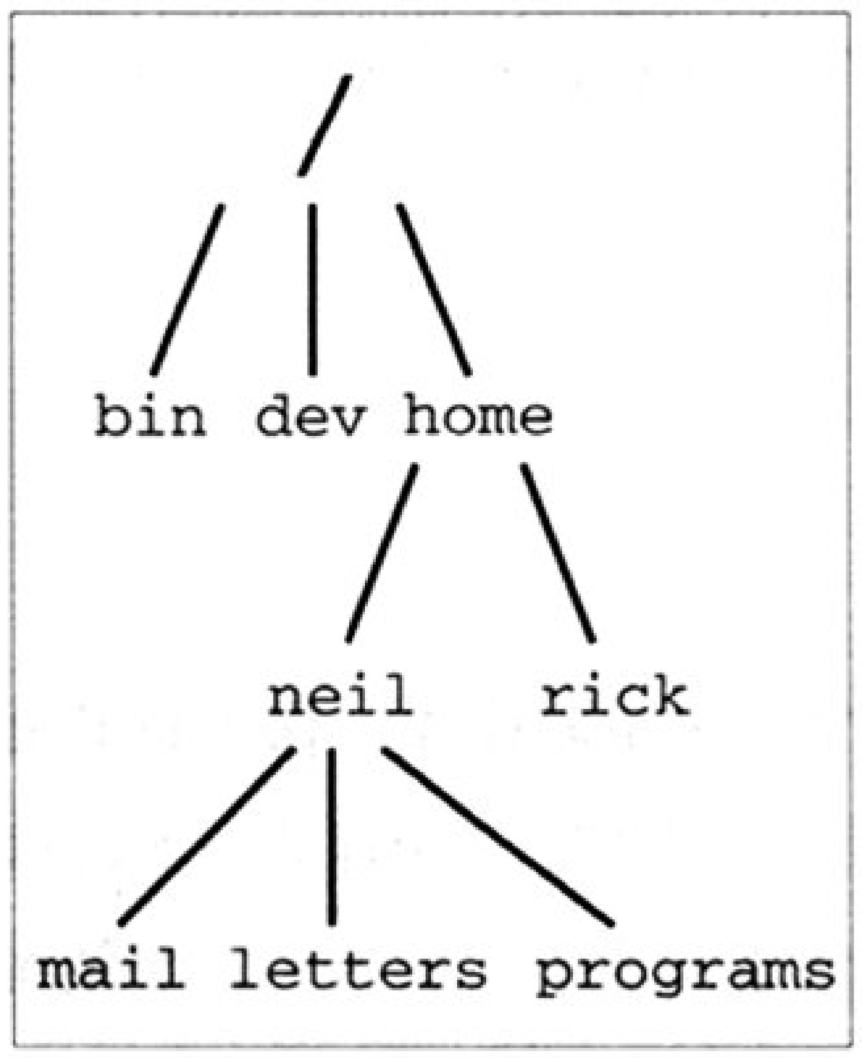


図3.1　ファイルシステムの階層構造

3.1.2 ファイルとデバイス

UNIXでは、ハードウェアデバイスさえもファイルによって表され（マップされ）ます。たとえば、CD-ROMドライブは、次のようにして**root**がファイルとしてマウントします。

```
# mount -rt iso9660 /dev/hdc /mnt/cd_rom
# cd /mnt/cd_rom
```

この例では、ブート時に**hdc**として認識されたCD-ROMデバイスの内容が、**/mnt/cd_rom**の下にファイル構造としてマウントされます。CD-ROMがいったんマウントされれば、いつもと同じようにその中のディレクトリを移動することができます。異なるのは、CD-ROMの内容が読み取り専用であるという点だけです。

次に、3つの重要な**デバイスファイル**について説明しておきましょう。

◆ /dev/console

このデバイスは、システムのコンソールを表します。エラーメッセージや診断情報は、しばしばこのデバイスに送られます。各UNIXシステムには、コンソールメッセージを受け取るための

専用の端末または画面があります。一般に、新しいワークステーションではアクティブな仮想コンソールが、Xでは画面上のコンソールウィンドウがこうしたメッセージを受け取ります。

◆ /dev/tty

スペシャルファイル`/dev/tty`は、プロセスが制御端末（キーボードと画面、またはウィンドウ）を持っている場合の制御端末のエイリアス（論理デバイス）です。たとえば、`cron`から実行されるプロセスは制御端末を持たないので、このようなプロセスは`/dev/tty`を開くことはできません。

`/dev/tty`を使用できる場合には、ユーザーがどの仮想端末またはハードウェア端末を使っているかに関係なく、このデバイスを使ってプログラムからユーザーに直接メッセージを送信することができます。この点については、第5章で詳しく取り上げます。

`/dev/console`は1つしかありませんが、`/dev/tty`を介してアクセスできる物理デバイスは実際には多数存在します。

3

◆ /dev/null

これは**ヌルデバイス**です。このデバイスに書き込まれた出力はすべて破棄されます。このデバイスを読み取ると、ただちにファイルの終端が返されるので、`cp`コマンドなどでは空のファイルとして利用することができます。不要な出力はしばしば`/dev/null`にリダイレクトされます。

> メモ
>
> `touch filename`コマンドを使って空のファイルを作成することもできます。このコマンドは、ファイルが存在する場合にはファイルの修正時刻を変更し、ファイルが存在しない場合には、指定された名前で新しいファイルを作成します。このコマンドを使ったときには、ファイルが存在する場合でも内容は空になりません。

```
$ echo do not want to see this >/dev/null
$ cp /dev/null empty_file
```

以上のほかにも、`/dev`ディレクトリには、ハードディスクやフロッピーディスク、通信ポート、テープドライブ、CD-ROM、サウンドカード、さらにシステムの内部状態を表すデバイスなど、さまざまなデバイスがあります。たとえば、`null`バイトを返す`/dev/zero`は、`null`バイトだけのファイルを作成するときに使います。デバイスによっては、スーパーユーザーのパーミッションがないとアクセスできないものもあります。たとえば、通常のユーザーは、ハードディスクなどの低水準デバイスに直接アクセスするプログラムを書くことはできません。デバイスファイルの名前は、システムによって異なることがあります。SolarisとLinuxには、通常のユーザーからはアクセスできないデバイスを管理するために、スーパーユーザーの権限で実行されるアプリケーションがあります。ユーザーがマウント可能なファイルシステムに対して使う`mount`コマンドなどは、その一例です。

この章では、特にディスクファイルとディレクトリについて解説しますが、第5章では別のデバイスとしてユーザー端末を取り上げます。

3.1.3 システムコールとデバイスドライバ

ファイルとデバイスは、ごくわずかな数の関数を使うだけでアクセスと管理を行うことができます。**システムコール**と呼ばれるこれらの関数は、UNIXによって直接提供され、オペレーティングシステム自体とのインタフェースとして機能します。

オペレーティングシステムの心臓部にあたるカーネルには、いくつかの**デバイスドライバ**があります。デバイスドライバは、システムハードウェアを制御するための低水準インタフェースの集合です。たとえば、テープドライブのデバイスドライバは、テープのスタート、送り戻し、読み書きなどの方法を知っています。また、テープに書き込むときにどのブロックサイズを使用するかについての情報も持っています。テープは本来シーケンシャルなものであり、ドライバはテープのブロックに直接アクセスすることはできないので、適切な位置までテープを送る必要があります。

一方、低水準ハードディスクデバイスドライバは、1回の書き込みをディスクのセクタ単位でしか行うことができません。しかし、任意のディスクブロックに直接アクセスすることができます。これは、ディスクがランダムアクセスデバイスだからです。

デバイスドライバは、どのデバイスに対しても同じようなインタフェースを提供するために、すべてのハードウェア依存機能をカプセル化しています。ハードウェアに特有の機能は、一般に`ioctl()`を通じて利用できるようになっています。

`/dev`ディレクトリ内のデバイスファイルは、すべて同じ方法で使われます。つまり、どのデバイスファイルも、オープンしたあと、読み取りや書き込みを行い、クローズするという方法で使用できます。たとえば、ディスクファイル、ユーザー端末、プリンタ、テープドライブのいずれに対しても、`open`を使ってアクセスすることができます。

デバイスドライバにアクセスするのに使われる低水準関数（システムコール）には、次のものがあります。

- `open`　ファイルまたはデバイスをオープンする。
- `read`　オープンされたファイルまたはデバイスから読み取る。
- `write`　ファイルまたはデバイスに書き込む。
- `close`　ファイルまたはデバイスをクローズする。
- `ioctl`　デバイスを制御する。

`ioctl`システムコールは、ハードウェア固有の制御を行うのに使われるため、使い方はデバイスごとに異なります。たとえば、テープドライブの巻き戻しに使うこともあれば、シリアルポー

トのフロー制御を設定するために使うこともあります。このため、**ioctl()**にはマシン間での移植性はありません。

上に挙げたものを含めて、システムコールのドキュメントは、マニュアルページのセクション2にあります。システムコールのパラメータリストと戻り値の型を記述したプロトタイプ、および関連する定数の**#define**文は、インクルードファイルにあります。各システムコールで必要になるインクルードファイルについては、以下の説明で具体的に示します。

3.1.4 ライブラリ関数

入出力のための低水準システムコールの問題は、あまり効率がよくないことです。

システムコールにはパフォーマンス上のペナルティが伴います。システムコールを使った場合、UNIXでは、プログラムコードの実行からカーネルコードの実行に切り替え、その後再びプログラムコードの実行に戻る必要があります。このため、関数呼び出しと比べてシステムコールは高価です。

ハードウェアにはさまざまな制限があり、この制限によって、低水準システムコールが一度に読み書きできるデータブロックのサイズも影響を受けます。たとえば、テープドライブに書き込める最小ブロックサイズには、10Kなどの制限があります。したがって、これより小さいサイズのデータを書き込む場合には、テープを10K分先送りすることになり、テープに無駄が生じます。

デバイスとディスクファイルに対する高水準インタフェースを提供するため、UNIXには一連の**標準ライブラリ**が用意されています。これらの標準ライブラリは、プログラムの中でさまざまな処理を行うために使用できる関数の集合です。たとえば、バッファ付き出力を提供するライブラリとして、**標準入出力ライブラリ**があります。標準入出力ライブラリを使えば、さまざまなサイズのデータブロックを効率的に書き込むことができます。ライブラリ関数は、十分な量のデータが揃ったときにフルブロックのデータを低水準システムコールに渡してくれるので、システムコールのオーバーヘッドは劇的に低下します。

通常、ライブラリ関数のドキュメントは、マニュアルページのセクション3にあります。一般に、ライブラリ関数には、その関数に関連付けられた標準インクルードファイルがあります。たとえば、標準入出力ライブラリの場合には、**stdio.h**がこれにあたります。

これまでに説明したことをまとめたのが次の図です。この図を見ながら、各種のファイル関数と、ユーザー、デバイスドライバ、カーネル、ハードウェアとの位置関係を確認してみてください。

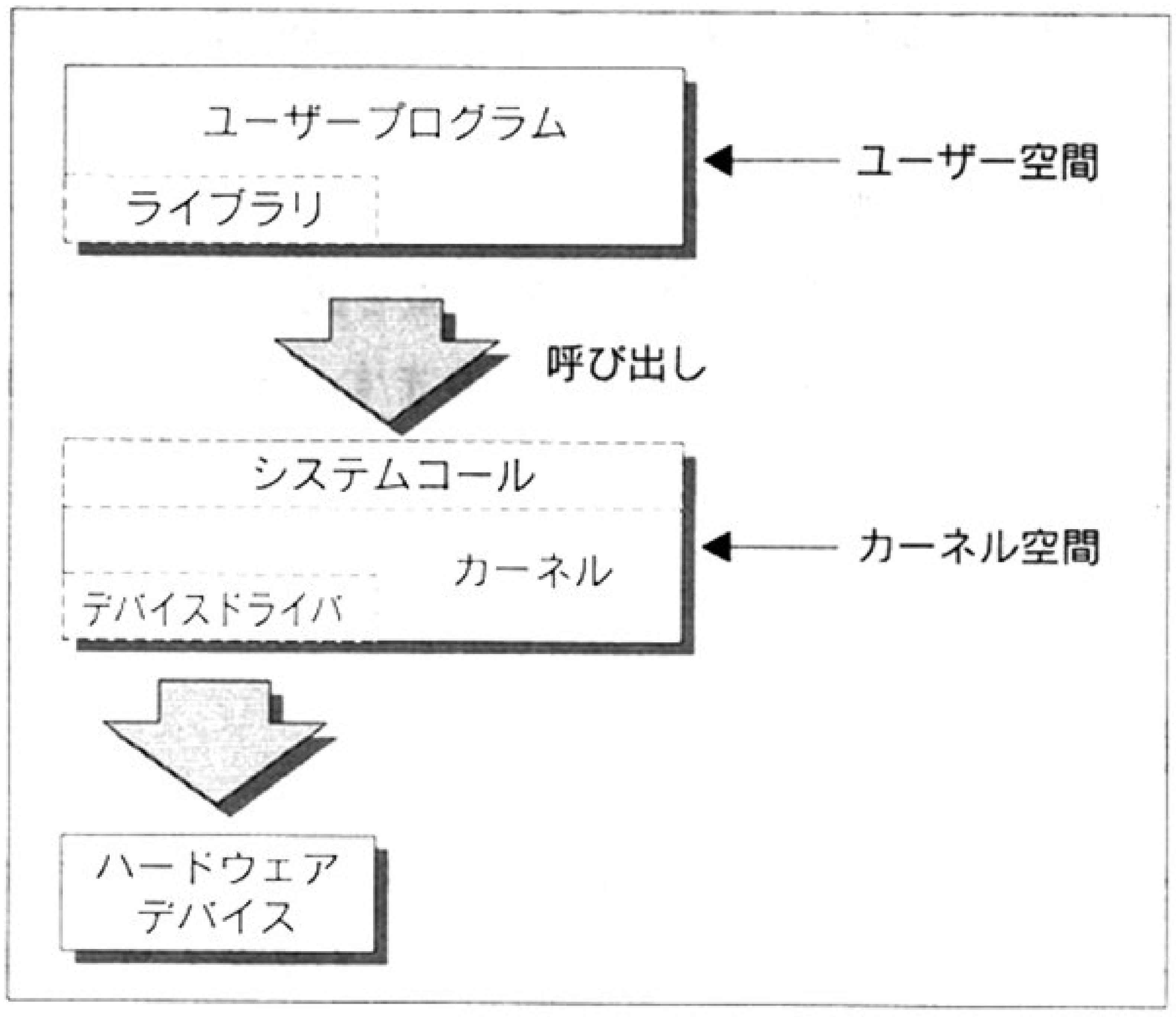


図3.2 UNIXシステムの位置関係

3.2 低水準ファイルアクセス

実行中の各プログラム（これを**プロセス**と呼びます）には、いくつかの**ファイルデスクリプタ**が関連付けられます。ファイルデスクリプタは、オープンしたファイルやデバイスにアクセスするときに使う小さな整数です。ファイルデスクリプタをいくつまで使用できるかは、UNIXシステムの構成によって異なります。通常、プログラムを実行すると、これらのファイルデスクリプタのうち、次の3つがすでにオープンされた状態になります。

- 0　　標準入力
- 1　　標準出力
- 2　　標準エラー

openシステムコールを使うと、上記以外のファイルデスクリプタもファイルやデバイスに関連付けることができます。自動的にオープンされるファイルデスクリプタを利用すれば、**write**を使ってごく簡単なプログラムを作成することもできます。

◆ write

```
#include <unistd.h>

size_t write(int fildes, const void *buf, size_t nbytes);
```

writeシステムコールは、ファイルデスクリプタfildesに関連付けられたファイルに、bufから最初のnbytesバイトを書き込み、実際に書き込まれたバイト数を返します。ファイルデスクリプタにエラーがあったり、デバイスドライバがブロックサイズの制約を受ける場合には、writeが返すバイト数がnbytesより小さくなることがあります。戻り値が0の場合には、ファイルの終わりに達しています。戻り値が-1の場合には、writeコールでエラーが発生したことを意味し、errnoにエラーの内容を示す値が設定されます。

次のsimple_write.cは、writeを使った簡単なプログラムです。

```
#include <unistd.h>

int main()
{
  if ((write(1,"Here is some data\n",18)) != 18)
    write(2,"A write error has occurred on file decriptor 1\n",47);

  exit(0);
}
```

これは、標準出力にメッセージを表示するだけのプログラムです。プログラムが終了すると、オープンされているすべてのファイルデスクリプタは自動的にクローズされるので、プログラムの中で明示的にファイルデスクリプタをクローズする必要はありません（ただし、バッファ付き出力の場合には明示的にクローズする必要があります）。

```
$ simple_write
Here is some data
$
```

注意

この章のすべての例は、PATHに現在のディレクトリが含まれていることを前提にしています。また、プログラムはスーパーユーザーではなく、通常のユーザーで実行してください。

◆ read

```
#include <unistd.h>

size_t read(int fildes, void *buf, size_t nbytes);
```

readシステムコールは、ファイルデスクリプタfildesに関連付けられたファイルから、nbytesバイトまでのデータをデータエリアbufに読み込み、実際に読み込んだバイト数を返します。返されるバイト数は、nbytesより小さくなることがあります。readが0を返した場合には、読み取るべきものがないこと、つまりファイルの終わりに達したことを意味します。エラー

が発生した場合には、-1が返されます。

次の**simple_read.c**プログラムは、標準入力の最初の128バイトを標準出力にコピーします。入力が128バイトより少ない場合には、入力のすべてがコピーされます。

```
#include <unistd.h>

int main()
{
    char buffer[128];
    int nread;

    nread = read(0, buffer, 128);

    if (nread == -1)
        write(2, "A read error has occurred\n", 26);

    if ((write(1,buffer,nread)) != nread)
        write(2, "A write error has occurred\n",27);

    exit(0);
}
```

このプログラムを実行すると、次のようになります。

```
$ echo hello there | simple_read
hello there
$ simple_read < draft1.txt
Files

In this chapter we will looking at files and directories and how to
manipulate them. We will learn how to create files, o$
```

シェルのプロンプト（**$**）が、最後に出力された行の途中に表示されていることに注意してください。これは、標準出力には128バイトしかコピーされず、これだけでは行が完結しないためです。

◆ open

新しいファイルデスクリプタを作成するには、**open**システムコールを使う必要があります。**creat**はPOSIXによって標準化されているもうひとつのオプションで、Linuxもサポートしていますが、現在ではほとんど使われません。

```
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

int open(const char *path, int oflags);
int open(const char *path, int oflags, mode_t mode);
```

> **メモ**
> 厳密にいうと、POSIXシステムでは`sys/types.h`と`sys/stat.h`をインクルードしなくても`open`を使用することができます。ただし、UNIXシステムによっては、これらのファイルをインクルードしなければならないことがあります。

簡単にいうと、**open**はファイルまたはデバイスへのアクセスパスを確立します。呼び出しが成功した場合には、ファイルデスクリプタを返します。返されたファイルデスクリプタは、**read**、**write**、およびその他のシステムコールで使用することができます。ファイルデスクリプタは一意であり、実行中のほかのプロセスとは共有されません。1つのファイルを2つのプログラムが同時にオープンした場合、これらのプログラムでは、それぞれ別個にファイルデスクリプタを使うことになります。両方のプログラムからファイルに書き込みを行う場合には、それぞれのプログラムが独自に保持している位置で書き込みが行われます。書き込まれるデータは交互にファイルに挿入されるのではなく、どちらかのプログラムのデータがもう一方のデータを上書きします。ファイル上の読み取りまたは書き込みの位置（オフセット）は、プログラムごとに管理されます。このため、2つのプログラムの間で衝突が起きないようにするには、ファイルのロックというしくみを利用する必要があります。ファイルのロックについては、第7章で取り上げます。

オープンするファイルまたはデバイスの名前は、パラメータ**path**で指定します。パラメータ**flags**には、ファイルをオープンするときの動作を指定します。

oflagsには、ファイルアクセスモード（省略不可）と、その他の省略可能オプションとのビット単位の論理和を指定します。**open**では、次のファイルアクセスモードのいずれかを指定しなければなりません。

表3.1 openのファイルアクセスモード

モード	説明
`O_RDONLY`	読み取り専用でオープン
`O_WRONLY`	書き込み専用でオープン
`O_RDWR`	読み書き用にオープン

ファイルアクセスモードとのビット単位の論理和によって**oflags**に指定できる省略可能なモードには、次のようなものがあります。

- **O_APPEND**　書き込まれるデータをファイルの最後に追加する。
- **O_TRUNC**　ファイルの長さを0に切り詰め、既存の内容を破棄する。
- **O_CREAT**　必要に応じて、**mode**で指定されたパーミッションでファイルを作成する。

- O_EXCL　O_CREATとともに指定された場合、すでにファイルが存在していればopenは失敗する。openはアトミック操作、つまりそれ以上分割できない1つの関数呼び出しとして処理される。このフラグは、ファイルを作成するのが呼び出し元であることを確実にする意味を持つ（2つのプログラムが同時にファイルの作成を試みた場合でも、どちらがファイルを作成したのかを判定できる）。

上記以外のoflagsのオプションについては、openのマニュアルページを参照してください。

成功した場合、openは新しいファイルデスクリプタを返します。失敗すると -1を返し、グローバル変数errnoにエラーの内容を示す値を設定します。errnoについては、あとで詳しく説明します。openは常に、未使用のファイルデスクリプタのうち最小のものを新しいファイルデスクリプタとして返します。このことを利用すると便利なことがあります。たとえば、プログラムの中で標準出力をクローズしたあと、openを呼び出すと、ファイルデスクリプタ1が再利用されることになり、標準出力を効率的に別のファイルやデバイスにリダイレクトすることができます。

◆ 初期パーミッション

openにO_CREATフラグを指定してファイルを作成する場合には、パラメータを3つ指定する必要があります。3番目のパラメータmodeには、ヘッダーファイルsys/stat.hで定義されているフラグのビット単位の論理和を指定します。指定できるフラグは次のとおりです。

- S_IRUSR　オーナーの読み取りパーミッション
- S_IWUSR　オーナーの書き込みパーミッション
- S_IXUSR　オーナーの実行パーミッション
- S_IRGRP　グループの読み取りパーミッション
- S_IWGRP　グループの書き込みパーミッション
- S_IXGRP　グループの実行パーミッション
- S_IROTH　その他のユーザーの読み取りパーミッション
- S_IWOTH　その他のユーザーの書き込みパーミッション
- S_IXOTH　その他のユーザーの実行パーミッション

たとえば、次のように指定します。

```
open ("myfile", O_CREAT, S_IRUSR|S_IXOTH);
```

この例では、オーナーが読み取り可能で、ほかのユーザーには実行可能なmyfileという名前のファイルが作成されます。

```
$ ls -ls myfile
   0 -r-------x   1 neil     software        0 Oct 30 08:20 myfile*
```

ファイルのパーミッションには、いくつかの要素が影響を与えます。まず、指定されたパーミッションは、ファイルが作成される場合にのみ使われます。次に、ユーザーマスク（シェルのumaskコマンドで指定します）も、作成されるファイルのパーミッションに影響します。具体的には、実行時に使われるモードの値は、openを呼び出すときに指定したモードの値とユーザーマスクの値を反転したものとの論理積になります。たとえば、ユーザーマスクが001になっていて、モードフラグにS_IXOTHが指定されている場合、作成されるファイルには、ほかのユーザーの実行パーミッションは設定されません。これは、001が、ほかのユーザーの実行パーミッションを許可しないユーザーマスクだからです。つまり、openとcreatで指定するモードフラグは、実際にはパーミッションの設定それ自体ではなく、あくまで設定の要求なのです。要求されたパーミッションが設定されるかどうかは、実行時のumaskの値に依存します。

◆ umask

umaskは、ファイルの作成時に使われるシステム変数で、ファイルパーミッションのマスクを表します。この変数の値を変更するには、新しい値を指定してumaskコマンドを実行します。値には、3桁の8進数を指定します。各桁は次の表に示すように、それぞれユーザー、グループ、ほかのユーザーのパーミッションに対応しており、値は1、2、4の論理和の結果を表します。

表3.2　umaskの値

桁	値	意味
1	0	ユーザーのパーミッションをいずれも許可
	4	ユーザーの読み取りパーミッションを不許可
	2	ユーザーの書き込みパーミッションを不許可
	1	ユーザーの実行パーミッションを不許可
2	0	グループのパーミッションをいずれも許可
	4	グループの読み取りパーミッションを不許可
	2	グループの書き込みパーミッションを不許可
	1	グループの実行パーミッションを不許可
3	0	ほかのユーザーのパーミッションをいずれも許可
	4	ほかのユーザーの読み取りパーミッションを不許可
	2	ほかのユーザーの書き込みパーミッションを不許可
	1	ほかのユーザーの実行パーミッションを不許可

たとえば、グループの書き込みと実行、およびほかのユーザーの書き込みをブロックするumaskは、次のようになります。

表3.3 umaskの値の例

桁	値
1	0
2	2
	1
3	2

各桁の値は、該当するすべての値の論理和になります。たとえば、2桁目の値は2と1の論理和で3になります。したがって、全体では、umaskの値は032になります。

openまたはcreatを呼び出してファイルを作成する場合には、パラメータmodeとumaskの値が比較されます。そして、パラメータmodeでセットされているビットのうち、umaskでもセットされているビットがクリアされます。このため、ユーザーの環境を適切に設定すれば、ファイルを作成するプログラムの中でほかのユーザーの書き込みパーミッションを要求していても、実際にそうしたパーミッションを持つファイルの作成を禁止することができます。ただし、これはファイルが作成される時点での話で、chmodコマンドを使えば、プログラムやユーザーがあとから書き込みパーミッションを追加することは可能です。umaskの存在意義は、新しいファイルのパーミッションを毎回ユーザーが設定したり、チェックしたりせずにすむという点にあります。

◆ close

```
#include <unistd.h>

int close(int fildes);
```

closeシステムコールは、ファイルデスクリプタfildesとファイルとの関連付けを終了するときに使います。関連付けがなくなったファイルデスクリプタは再利用可能になります。成功した場合には0を返し、エラーが発生した場合には-1を返します。

実行中の1つのプログラムが、オープンされたファイルを同時にいくつまで持てるかについては制限が設けられています。この値は、limits.hで定数OPEN_MAXによって定義されています。値はシステムによって異なりますが、POSIXでは少なくとも16でなければならないとしています。ただしこれ自体、ローカルシステムの制限によって制約を受けることがあります。

◆ ioctl

```
#include <unistd.h>

int ioctl(int fildes, int cmd, ...);
```

ioctlはいってみればなんでも屋的な関数で、デバイスとそのデスクリプタのふるまいを制御

したり、デバイスが提供するサービスを設定したりするためのインタフェースを提供します。端末、ファイルデスクリプタ、ソケット、テープドライブなど、さまざまなデバイスでそれぞれ独自の**ioctl**の呼び出しが定義されており、その詳しい内容については該当するデバイスのマニュアルページを参照する必要があります。POSIXでは、ストリーム用の**ioctl**を定義しているだけです（本書では触れません）。

ioctlは、デスクリプタ**fildes**で指定されたオブジェクトに対して、**cmd**で指定された機能を実行します。デバイスがどのような機能をサポートしているかによって、3番目の引数を取ることもあります。

例題 ファイルコピープログラム

open、**read**、**write**システムコールについてひととおり説明を終えたので、ファイルの内容を1文字ずつコピーする**copy_system.c**というプログラムを作成してみましょう。

> **注意**
> この章では、これ以降、同じプログラムをさまざまな方法で作成し、実行効率がどのように変化するかを実際に確かめていきます。サンプルプログラムの記述は、入力ファイルがすでに存在し、出力ファイルは存在しないことを前提としていますが、これは単にコードが長くなるのを避けるためです。実際にプログラムを作成する場合には、ファイルが存在するかどうかなどのチェックをきちんと行う必要があります。

```
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>

int main()
{
    char c;
    int in, out;

    in = open("file.in", O_RDONLY);
    out = open("file.out", O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR);
    while(read(in, &c, 1) == 1)
        write(out, &c, 1);

    exit(0);
}
```

> **注意**
> #include <unistd.h>の行は先頭に置く必要があります。これは、ほかのインクルードファイルに影響するPOSIX準拠のフラグがこのファイルで定義されているためです。

このプログラムを実行すると、次のようになります。

```
$ time copy_system
1.09user 20.18system 0:21.27elapsed 100%CPU (0avgtext+0avgdata
0maxresident)k
...
$ ls -ls file.in file.out
1029 -rw-r--r--   1 neil     users     1048576 Oct 30 15:10 file.in
1029 -rw-------   1 neil     users     1048576 Oct 30 15:12 file.out
```

上の実行例では、timeを使ってプログラムの実行にかかった時間を計っています。1MBの入力ファイルfile.inは、オーナーだけが読み取りと書き込みのパーミッションを持つファイルとして作成されたfile.outに正しくコピーされています。ただし、コピーには20秒ほどかかっており、実質的にCPU時間のすべてを消費しています。これは、合計で200万回以上のシステムコールを行っていることが原因です。

もっと大きなかたまりごとにコピーを行えば、実行速度を改善することができます。たとえば、次のcoyp_block.cでは、1Kのブロックごとにファイルをコピーしています。

```
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>

int main()
{
    char block[1024];
    int in, out;
    int nread;

    in = open("file.in", O_RDONLY);
    out = open("file.out", O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR);
    while((nread = read(in, block, sizeof(block))) > 0)
        write(out, block, nread);

    exit(0);
}
```

次に、出力ファイルを削除してからプログラムを実行します。

```
$ rm file.out
$ time copy_block
0.00user 0.06system 0:00.14elapsed 40%CPU (0avgtext+0avgdata
0maxresident)k
...
$ ls -ls file.in file.out
1029 -rw-r--r--   1 neil     users     1048576 Oct 30 15:10 file.in
1029 -rw-------   1 neil     users     1048576 Oct 30 16:44 file.out
```

今度は1秒もかからずにプログラムを実行できました。これは、システムコールが約2000回ですんでいるからです。実行時間の具体的な値はシステムによって異なりますが、上の2つの例の比較から、システムコールのオーバーヘッドがかなり大きいことがわかります。つまり、システムコールの使い方を最適化すれば、実行速度を改善できるのです。

3.2.1 ファイル操作のためのその他のシステムコール

これまでに説明したもののほかにも、低水準ファイルデスクリプタを操作するシステムコールがいくつか存在します。これらのシステムコールを使うと、ファイルの使い方を制御したり、ステータス情報を取得したりすることができます。ここでは、これらのシステムコールをまとめて解説しておきます。必要に応じてあとから読み返すとよいでしょう。

◆ lseek

```
#include <unistd.h>
#include <sys/types.h>

off_t lseek(int fildes, off_t offset, int whence);
```

lseekシステムコールは、ファイルデスクリプタ**fildes**の読み取り/書き込みポインタをセットします。**lseek**を使えば、次にファイルの読み取りや書き込みを行う場所を設定することができます。ポインタは、ファイル内の絶対位置か、現在の位置またはファイルの最後からの相対位置として設定できます。パラメータ**offset**には位置を指定し、パラメータ**whence**にはオフセットの意味を指定します。**whence**には次のいずれかを指定できます。

- **SEEK_SET**　　**offset**は絶対位置
- **SEEK_CUR**　　**offset**は現在の位置からの相対位置
- **SEEK_END**　　**offset**はファイルの最後からの相対位置

lseekは、ファイルポインタの設定されているオフセットをファイルの先頭からのバイト数で返します。失敗した場合には -1を返します。シーク操作の**offset**で使われる**off_t**型は、**sys/types.h**で定義されている処理系依存の型です。

◆ fstat、stat、lstat

```
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>


int fstat(int fildes, struct stat *buf);
int stat(const char *path, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

fstatシステムコールは、オープンされたファイルデスクリプタに関連付けられているファイルのステータス情報を返します。ステータス情報は、構造体**buf**に書き込まれます。このシステムコールを呼び出すときには、構造体**buf**のアドレスをパラメータとして渡します。

残りの2つの関数**stat**と**lstat**は、指定された名前のファイルに関するステータス情報を返します。どちらも同じ結果を返しますが、ファイルがシンボリックリンクの場合には動作が異なります。**lstat**はリンクそれ自体に関する情報を返しますが、**stat**はリンクが指しているファイルの情報を返します。

構造体**stat**のメンバはUNIXシステムによって異なりますが、必ず次のものが含まれています。

表3.4 stat構造体のメンバ

statのメンバ	説明
st_mode	ファイルのパーミッションと種類に関する情報
st_ino	ファイルに関連付けられているiノード
st_dev	ファイルが存在するデバイス
st_uid	ファイルのオーナーのユーザーID
st_gid	ファイルのオーナーのグループID
st_atime	最終アクセス時刻
st_ctime	モード、オーナー、グループ、または内容の最終変更時刻
st_mtime	内容の最終修正時刻
st_nlink	ファイルへのハードリンク数

stat構造体に返される**st_mode**フラグについては、**sys/stat.h**でいくつかのマクロが定義されています。これらのマクロは、パーミッションとファイルの種類に関するフラグ、および特定の種類とパーミッションを調べるためのマスクとして使われます。

パーミッションフラグは、すでに説明した**open**システムコールのものと同じです。ファイルの種類を表すフラグには次のものがあります。

- **S_IFBLK** ブロックスペシャルデバイス
- **S_IFDIR** ディレクトリ

- S_IFCHR　キャラクタスペシャルデバイス
- S_IFIFO　FIFO（名前付きパイプ）
- S_IFREG　通常のファイル
- S_IFLNK　シンボリックリンク

その他のフラグには次のものがあります。

- S_ISUID　setUIDがセットされている。
- S_IGUID　setGIDがセットされている。

st_modeフラグを解釈するためのマスクには次のものがあります。

- S_IFMT　ファイルの種類
- S_IRWXU　ユーザーの読み取り／書き込み／実行パーミッション
- S_IRWXG　グループの読み取り／書き込み／実行パーミッション
- S_IRWXO　その他のユーザーの読み取り／書き込み／実行パーミッション

ファイルの種類を調べるためのフラグには次のものがあります。これらのマクロは、適切な方法でマスクされたモードフラグと、デバイスの種類を表すフラグとを比較します。

- S_ISBLK　ブロックスペシャルファイルかどうかを調べる。
- S_ISCHR　キャラクタスペシャルファイルかどうかを調べる。
- S_ISDIR　ディレクトリかどうかを調べる。
- S_ISFIFO　FIFOかどうかを調べる。
- S_ISREG　通常のファイルかどうかを調べる。
- S_ISLNK　シンボリックリンクかどうかを調べる。

たとえば、ファイルがディレクトリではなく、オーナーのパーミッションのうち実行パーミッションだけがセットされていることを調べるには、次のようにします。

```
struct stat statbuf;
mode_t modes;

stat("filename",&statbuf);
modes = statbuf.st_mode;

if(!S_ISDIR(modes) && (modes & S_IRWXU) == S_IXUSR)
    ...
```

◆ dup、dup2

```
#include <unistd.h>

int dup(int fildes);
int dup2(int fildes, int fildes2);
```

dupシステムコールは、ファイルデスクリプタを複製する手段を提供し、同じファイルに異なる複数のファイルデスクリプタでアクセスできるようにします。これらのファイルデスクリプタは、同じファイルの別の場所を読み書きするときに使用できます。**dup**はファイルデスクリプタ**fildes**を複製し、新しいデスクリプタを返します。

dup2システムコールはファイルデスクリプタをコピーします。**fildes2**は、コピー先の新しいファイルデスクリプタとして使われます。

これらのシステムコールは、パイプを介して通信する複数のプロセスを使う場合にも便利です。**dup**システムコールについては、第11章でもう一度取り上げます。

3.3 標準入出力ライブラリ

標準入出力ライブラリとそのヘッダーファイル**stdio.h**は、低水準入出力システムコールに対する柔軟なインタフェースを提供し、書式付き出力や入力の走査を行うための便利な関数を数多く用意しています。また、このライブラリは、デバイスのバッファリングも行います。

標準入出力ライブラリは、低水準ファイルデスクリプタと同じ方法で使います。具体的には、まず、ファイルをオープンしてアクセスパスを確立する必要があります。ファイルをオープンしたときに返される値は、ほかの入出力ライブラリ関数のパラメータとして使います。標準入出力ライブラリで低水準ファイルデスクリプタに相当するものは、**ストリーム**と呼ばれ、構造体へのポインタ**FILE ***として実装されています。

メモ

ここで説明しているストリームと、AT&T UNIX System V Release 3で導入されたプロセス間通信の**STREAMS**（本書では取り上げません）とは別のものです。**STREAMS**について詳しく知りたい場合には、X/Open仕様およびSystem V付属のAT&T **STREAMS**プログラミングガイドを参照してください。

プログラムを実行すると、**stdin**、**stdout**、**stderr**の3つのファイルストリームが自動的にオープンされます。**stdin**、**stdout**、**stderr**は**stdio.h**で宣言されており、それぞれ標準入力、標準出力、標準エラー出力を表します。これら3つのファイルストリームは、低水準ファイルデスクリプタの0、1、2に対応します。

以下では、標準入出力に含まれる次のような関数について説明していきます。

- fopen、fclose
- fread、fwrite
- fflush
- fseek
- fgetc、getc、getchar
- fputc、putc、putchar
- fgets、gets
- printf、fprintf、sprintf
- scanf、fscanf、sscanf

◆ fopen

```
#include <stdio.h>

FILE *fopen(const char *filename, const char *mode);
```

fopenライブラリ関数は、低水準**open**システムコールに似た機能を持ち、主にファイルや端末入出力に使います。デバイスを細かく制御する必要がある場合には、入出力バッファリングなど、ライブラリによる副作用を伴わない低水準システムコールのほうが適しています。

fopenは、パラメータ**filename**に指定された名前のファイルをオープンし、このファイルにストリームを関連付けます。**mode**は、ファイルをオープンする方法を指定するパラメータで、次のいずれかを指定します。

- **"r"**または**"rb"**　読み取り専用でオープンする。
- **"w"**または**"wb"**　書き込み用にオープンし、もとのファイルの長さを0に切り詰める。
- **"a"**または**"ab"**　書き込み用にオープンし、ファイルの最後に追加する。
- **"r+"**または**"rb+"**または**"r+b"**　更新（読み書き）用にオープンする。
- **"w+"**または**"wb+"**または**"w+b"**　更新用にオープンし、もとのファイルの長さを0に切り詰める。
- **"a+"**または**"ab+"**または**"a+b"**　更新用にオープンし、ファイルの最後に追加する。

bは、ファイルがテキストファイルではなくバイナリファイルであることを示します。ただし、

DOSとは異なり、UNIXではテキストファイルとバイナリファイルの区別はありません。UNIXではすべてのファイルを同じように扱い、いずれもバイナリファイルとして処理します。なお、パラメータ**mode**には、文字ではなく文字列を指定することに注意してください。たとえば、**'r'**ではなく**"r"**のように指定します。

成功すると、**fopen**はヌルではない**FILE ***ポインタを返します。失敗した場合には、**stdio.h**で定義されている値**NULL**を返します。

◆ fread

```
#include <stdio.h>

size_t fread(void *ptr, size_t size, size_t nitems, FILE *stream);
```

freadライブラリ関数は、ファイルストリームからデータを読み込むときに使います。データはストリーム**stream**から、**ptr**が示すデータバッファに読み込まれます。**fread**と次の**fwrite**は、データレコードを扱うときに便利な関数です。データレコードの指定は、レコードサイズ**size**と転送するレコード数**nitems**によって行います。**fread**は、データバッファへの読み込みに成功したアイテムの数（バイト数ではありません）を返します。したがって、ファイルの最後に達すると、**nitems**より小さい値が返されることになります。この**fread**を含めて、バッファへの書き込みを行う標準入出力関数では、プログラマがデータ用の領域を確保する必要があります。

◆ fwrite

```
#include <stdio.h>

size_t fwrite(void *ptr, size_t size, size_t nitems, FILE *stream);
```

fwriteライブラリ関数は、**fread**と似たインタフェースを持ち、指定されたデータバッファからデータレコードを出力ストリームに書き込みます。**fwrite**は、書き込みに成功したレコード数を返します。

注意

> 構造化されたデータを異なるシステム間で転送する場合、不用意に**fread**や**fwrite**を使うと問題が起きることがあります。

◆ fclose

```
#include <stdio.h>

int fclose(FILE *stream);
```

fcloseライブラリ関数は、指定された**stream**をクローズし、まだ書き込まれていないデータを書き込みます。標準入出力ライブラリではデータがバッファリングされるので、**fclose**の使用は重要な意味を持っています。データが最後まで完全に書き込まれたことをプログラムで保証する必要がある場合には、**fclose**を呼び出してストリームをクローズする必要があります。ただし、プログラムが正常に終了した場合には、その時点でまだオープンされているすべてのファイルストリームに対して**fclose**が自動的に呼び出されます。ファイルデスクリプタの場合と同様、使用できるストリームの数は制限されています。この値は、**stdio.h**で定数**FOPEN_MAX**によって定義されています（最低8）。

◆ fflush

```
#include <stdio.h>

int fflush(FILE *stream);
```

fflushライブラリ関数は、ファイルストリームにあるすべての未処理データをただちに書き込みます。たとえば、ユーザーと対話的なやりとりを行うプログラムの中で**fflush**を使うと、プロンプトを端末に確実に送り出したあと、ユーザーの入力を読み取ることができます。また、重要なデータをいったんディスクにコミットしてから処理を続けるといったことも可能です。プログラムをデバッグする場合にも、要所要所で**fflush**を使えば、データがきちんと書き込まれているかどうかを確認することができます。**fclose**が呼び出されると、ストリームは暗黙のうちにフラッシュされます。したがって、**fclose**の前に**fflush**を呼び出す必要はありません。

◆ fseek

```
#include <stdio.h>

int fseek(FILE *sream, long int offset, int whence);
```

fseek関数は**lseek**システムコールのファイルストリーム版で、ストリーム上で次に読み書きを行う位置を設定します。パラメータ**offset**と**whence**の意味と値は、**lseek**の場合と同じです。ただし、**lseek**は**off_t**型を返しますが、**fseek**は成功した場合、0を返します。失敗した場合には-1を返し、**errno**にエラーの内容を示す値を設定します。

◆ fgetc、getc、getchar

```
#include <stdio.h>

int fgetc(FILE *stream);
int getc(FILE *stream);
int getchar();
```

fgetc関数は、ファイルストリームから次のバイトを文字として返します。ファイルの最後に達すると、**EOF**を返します。

getc関数は、マクロとして実装できる点を除けば**fgetc**と同じです。**getc**がマクロになっている場合、副作用のあるものを引数**stream**に指定することはできません。また、**getc**を関数のポインタとして使うこともできません。

getchar関数は**getc(stdin)**と等価で、標準入力から次の文字を読み取ります。

◆ fputc、putc、putchar

```
#include <stdio.h>

int fputc(int c, FILE *stream);
int putc(int c, FILE *stream);
int putchar(int c);
```

fputc関数は出力ファイルストリームに1文字を書き込み、書き込んだ文字を返します。エラーが発生した場合には、**EOF**を返します。

fgetcと**getc**の場合と同様、マクロとして実装できる点を除けば、**putc**は**fputc**と同じです。

putchar関数は**putc(c, stdout)**と等価で、標準出力に1文字を書き込みます。**putchar**は、**char**型ではなく**int**型を引数に取る点に注意してください。また、戻り値の型は**int**なので、ファイルの終わりを表すインジケータ（**EOF**）が値 -1を取ることができ、通常の文字との区別が可能になっています。

◆ fgets、gets

```
#include <stdio.h>

char *fgets(char *s, int n, FILE *stream);
char *gets(char *s);
```

fgets関数は、入力ファイル**stream**から文字列を読み込みます。読み込んだ文字は、改行文字に出合うか、**n-1**文字が転送されるか、またはファイルの終わりに達するかのいずれかの条件

が満たされるまで、sで指定されたバッファに書き込まれます。改行文字に出合った場合には改行文字もバッファに書き込まれ、そのあとに文字列の終わりを示す\0が追加されます。1回の呼び出しで読み取られる文字数は、最大n-1文字です。これは、バッファの最後に、文字列の終わりを示す\0を追加する必要があるからです（この\0も含めると、全体で最大nバイトになります）。

fgetsは、成功するとバッファへのポインタsを返します。ストリームがファイルの終わりに達した場合には、EOFインジケータをセットし、ヌルポインタを返します。読み取りエラーが起きた場合、fgetsはヌルポインタを返し、errnoにエラーの内容を示す値を設定します。

gets関数は、標準入力から読み取りを行う点と、改行文字を破棄する点を除けばfgetsと同じです。ただし、getsは転送できる文字数を制限していないので、転送バッファがあふれる可能性があります。このため、通常はgetsではなく、fgetsを使うべきです。

3.3.1 書式付き入出力

標準入出力ライブラリには、指定した形式で出力を生成するライブラリ関数も用意されています。ファイルストリームに値を書き込むprintfとその系列の関数、ファイルストリームから値を読み取るscanfなどがあります。

◆ printf、fprintf、sprintf

```
#include <stdio.h>

int printf(const char *format, ...);
int sprintf(char *s, const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
```

printf族の関数は、さまざまな型の可変個の引数に書式を設定して出力します。出力ストリーム中での各変数の表現方法は、パラメータformatで制御します。formatは、出力ストリームにそのまま表示される通常の文字と、formatに続く引数の表示方法および表示場所を指定するための**変換指定子**と呼ばれるコードから構成されます。

printf関数は出力を標準出力に書き込みます。fprintf関数は、出力先がstreamで指定されたストリームになる点を除けば、printfと同じです。sprintf関数は、文字列の終わりを示す\0を出力に付け、パラメータsで指定された文字列に結果を書き込みます。文字列sは、出力のすべてを受け取るだけの十分な大きさがなければなりません。

書式指定文字列formatの中の通常の文字は、そのまま出力されます。変換指定子がある場合には、パラメータとして渡された引数を順に取り出して書式を設定します。変換指定子は、常に文字%で始まります。次の例を見てください。

```
printf("Some numbers: %d, %d, and %d\n", 1, 2, 3);
```

このコードを実行すると、次のように出力されます。

```
Some numbers: 1, 2, and 3
```

文字%そのものを出力するには、変換指定子と区別するために%%と指定する必要があります。次に示すのは、よく使われる変換指定子です。

- %d、%i 整数を10進数で出力
- %o、%x 整数を8進数、16進数で出力
- %c 文字を出力
- %s 文字列を出力
- %f 浮動小数点数（単精度）を出力
- %e 倍精度浮動小数点数を出力
- %g 一般的な形式でdoubleを出力

printfに渡される引数の数と型は、書式指定文字列の変換指定子の数および型と一致している必要があります。整数引数の型を指定するには、オプションのサイズ指定子を使います。たとえば、h（%hdのように指定）はshort int、lはlong intを表します。コンパイラによっては、printfステートメントで使われている型などをチェックできるものがありますが、完全なものではありません。

次の例を見てください。

```
char initial = 'A';
char *surname = "Matthew";
double age = 6.5;

printf("Hello Miss %c %s, aged %g\n", initial, surname, age);
```

このコードを実行すると、次のように出力されます。

```
Hello Miss A Matthew, aged 6.5
```

フィールド指定子を使うと、出力をさらに細かく制御することができます。フィールド指定子は変換指定子に加えて使用し、出力中の文字揃えや空白の埋め込み方法を指定するものです。一般的には、浮動小数点数の小数部の桁数を指定したり、文字列の前後に挿入する空白を指定したりするのに使います。

フィールド指定子は、変換指定子の文字%のすぐあとに数値として指定します。次の表は、さ

まざまなフィールド指定子の使い方と、その出力結果を示したものです（出力位置をわかりやすくするために、出力を縦線で囲んでいます）。

表3.5　フィールド指定子

<table>
<tr><th>書式</th><th>引数</th><th>出力</th></tr>
<tr><td><code>%10s</code></td><td><code>"Hello"</code></td><td><code>| Hello|</code></td></tr>
<tr><td><code>%-10s</code></td><td><code>"Hello"</code></td><td><code>|Hello |</code></td></tr>
<tr><td><code>%10d</code></td><td><code>1234</code></td><td><code>| 1234|</code></td></tr>
<tr><td><code>%-10d</code></td><td><code>1234</code></td><td><code>|1234 |</code></td></tr>
<tr><td><code>%010d</code></td><td><code>1234</code></td><td><code>|0000001234|</code></td></tr>
<tr><td><code>%10.4f</code></td><td><code>12.34</code></td><td><code>| 12.3400|</code></td></tr>
<tr><td><code>%*s</code></td><td><code>10, "Hello"</code></td><td><code>| Hello|</code></td></tr>
</table>

表に示した例は、いずれも10文字分のフィールド幅で出力されています。2番目の例のようにフィールド幅に負の値を指定すると、出力は左寄せされます。また、最後の例のようにアスタリスク（`*`）を使えば、可変フィールド幅を指定することができます（表の例では、実際の幅には引数の10が使われています）。`0`（ゼロ）を前に付けた場合には、出力される項目の先行部分にゼロが埋め込まれます。POSIX仕様によれば、**`printf`** はフィールドをあふれさせることはせず、出力結果が収まるようにフィールドを拡張します。

`printf` 族の関数は、書き込まれた文字数を返します。**`sprintf`** では、最後の`\0`は文字数に含まれません。エラーが発生した場合には負の値を返し、**`errno`** を設定します。

◆ scanf、fscanf、sscanf

```
#include <stdio.h>

int scanf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int sscanf(const char *s, const char *format, ...);
```

`scanf` 族の関数は、**`printf`** 族と同じように動作しますが、ストリームから値を読み取り、読み取った値を、ポインタパラメータで渡されたアドレスの変数に格納する点が異なります。入力変換の方法を指定する書式指定文字列の使い方は **`printf`** 族と同じで、変換指定子もほとんど同じものを使います。

`scanf` 族では、**`printf`** 族と同様、入力された値を格納するのに使う変数には適切な型を使う必要があり、また渡される引数の数と型も、書式指定文字列中の変換指定子の数および型と一致している必要があります。一致していない場合には、メモリの内容が破壊され、プログラムがクラッシュすることがあります。このような場合でも、コンパイラからのエラーは表示されません。ただし、コンパイラによっては警告を表示することがあります。

scanf族の書式指定文字列には、printfの場合と同様、通常の文字と変換指定子の両方が含まれています。ただし、scanf族では、通常の文字は入力中に存在していなければならない文字を指定するのに使います。

次の例を見てください。

```
int num;

scanf("Hello %d", &num);
```

この例では、標準入力からの次の5文字がHelloに一致する場合にのみ、scanfの呼び出しが成功します。そして、この5文字の次の文字が10進数として認識できる値であれば、その値が読み取られて変数numに代入されます。書式指定文字列中の空白は、入力中のすべてのホワイトスペース（空白、タブ、フォームフィード、改行）を無視するために使います。したがって、次の2とおりの入力を与えた場合には、いずれもscanfの呼び出しが成功し、値1234が変数numに格納されることになります。

```
Hello      1234
Hello1234
```

通常、入力中のホワイトスペースは、変換が開始された時点でも無視されます。このため、上の書式指定文字列中の%dは、数字に出合うまで、スペースや改行をスキップしながら入力の読み取りを続けます。期待した文字が現れない場合、入力の変換は失敗します。

変換指定子には次のようなものがあります。

- %d　整数を10進数で入力
- %o、%x　整数を8進数、16進数で入力
- %f、%e、%g　浮動小数点数を入力
- %c　文字を入力（ホワイトスペースはスキップされない）
- %s　文字列を入力
- %[]　文字の集合を入力（後述）
- %%　文字%%を入力

printfと同様、scanfの変換指定子には、入力の文字数を制限するフィールド幅を指定することができます。サイズ指定子（shortを意味するhまたはlongを意味するlのいずれか）は、受け取る引数がデフォルトより短いか長いかを示すものです。たとえば、%hdはshort int、%ldはlong int、%lgは倍精度浮動小数点数を表します。

入力から単一の文字を読み取るには変換指定子%cを使います。%cを指定した場合、先行するホワイトスペースの読み飛ばしは行われません。

文字列を入力するには変換指定子%sを使います。ただし、%sを使うときには注意が必要です。先行するホワイトスペースはスキップされますが、文字列中で最初にホワイトスペースが出現した時点で読み取りが打ち切られるからです。このため、%sは一般的な文字列ではなく、単語を読み取るのに適しています。また、フィールド幅の指定がない場合、読み取る文字列の長さは制限されません。したがって、入力を格納する文字列は十分な長さを持っていなければなりません。通常は、フィールド指定子を使うか、またはfgetsとsscanfを組み合わせて使用し、入力行全体を読み取ったあとに走査するほうが無難です。

変換指定子%[]は、指定された文字の集合からなる文字列を読み取ります。たとえば、%[A-Z]と指定すると、大文字からなる文字列を読み取ります。指定された文字の最初の文字がキャレット（^）の場合には、指定された集合に含まれない文字からなる文字列を読み取ります。たとえば、スペースを含む文字列を読み取り、最初のカンマが出現した時点で読み取りを終了するには、%[^,]と指定します。

たとえば、次のような入力行があるとします。

```
Hello, 1234, 5.678, X, string to the end of the line
```

この行を次のコードで読み取ると、4つの項目が正しく入力されます。

```
char s[256];
int n;
float f;
char c;

scanf("Hello,%d,%g, %c, %[^\n]", &n,&f,&c,s);
```

scanf族の関数は、読み取った項目の数を返します。最初の項目で失敗した場合には0を返します。最初の項目を読み取る前に入力行の終わりに達した場合には、EOFを返します。ファイルストリーム上で読み取りエラーが発生した場合には、ストリームのエラーフラグがセットされ、errnoにエラーの内容を示す値が設定されます。ストリームのエラーについては、すぐあとで取り上げます。

一般に、次の3つの理由からscanf族の使用は勧められません。

- 実装には伝統的にバグが多い。
- 柔軟な使い方ができない。
- なにを解析しているのかわかりにくいコードになる。

3.3.2 その他のストリーム関数

これまでに取り上げたもののほかにも、次のようなさまざまな標準入出力ライブラリ関数があります。

- **fgetpos** ファイルストリーム中の現在の位置を取得する。
- **fsetpos** ファイルストリーム中の現在の位置を設定する。
- **ftell** ストリーム中の現在のファイルオフセットを返す。
- **rewind** ストリーム中のファイル位置をリセットする。
- **freopen** ファイルストリームを再利用する。
- **setvbuf** ストリームのバッファリング方式を設定する。
- **remove** パラメータ**path**がディレクトリでない場合には**unlink**と同じで、ディレクトリの場合には**rmdir**と同じ。

これらのライブラリ関数のドキュメントは、マニュアルページのセクション3にあります。

ライブラリ関数の説明がひととおり済んだので、この章で最初に作成したファイルコピープログラムをファイルストリーム関数を使って書き直してみましょう。ファイル名は**copy_stdio.c**とします。

例題 ファイルコピープログラム

これまでに作成したバージョンと似ていますが、ここでは標準入出力ライブラリ関数を使って1文字単位でファイルをコピーします。

```
#include <stdio.h>

int main()
{
  int c;
  FILE *in, *out;

  in = fopen("file.in", "r");
  out = fopen("file.out", "w");

  while((c = fgetc(in)) != EOF)
    fputc(c, out);

  exit(0);
}
```

これまでと同じ方法で実行します。

```
$ time copy_stdio
0.52user 0.00system 0:00.61elapsed 84%CPU (0avgtext+0avgdata
0maxresident)k
```

解説

ブロック単位でコピーした場合ほど速くはありませんが、低水準入出力関数で1文字ごとにコピーした場合と比べると、はるかに高速に処理できることがわかります。これは、標準入出力ライブラリでは**FILE**構造体の内部にバッファがあり、このバッファがいっぱいになったときに初めて低水準システムコールが呼び出されるためです。上の例では1文字単位でファイルをコピーしていますが、標準入出力ライブラリを使って行ごと、ブロックごとにコピーすることも可能ですから、ぜひ自分でいろいろと試してみてください。

3.3.3 ストリームのエラー

標準入出力ライブラリの多くの関数は、エラーを示すためにヌルポインタや定数**EOF**など、範囲外の値を返します。エラーが発生した場合には、エラーの内容を示す値が外部変数**errno**に設定されます。

```
#include <errno.h>

extern int errno;
```

注意

errnoの値を変更する関数は数多く存在します。したがって、**errno**の値は、関数でエラーが発生した時点でのみ有効です。値を調べたい場合には、関数からエラーが返された直後に調べる必要があります。また、**errno**の値を使う場合には、使用する前にほかの変数に値をコピーしておく必要があります。これは、値を表示するときに使う**fprintf**などの関数によって、**errno**の値が書き換えられる可能性があるからです。

次に示す関数を使ってファイルストリームの状態を調べると、エラーが発生しているかどうか、ファイルの終わりに達しているかどうかなどを知ることができます。

```
#include <stdio.h>

int ferror(FILE *stream);
int feof(FILE *stream);
void clearerr(FILE *stream);
```

ferror関数は、ストリームのエラーインジケータを調べ、エラーインジケータがセットされ

ている場合には0以外の値を、そうでない場合には0を返します。

feof関数は、ストリームの**EOF**インジケータを調べ、**EOF**インジケータがセットされている場合には0以外の値を、そうでない場合には0を返します。たとえば、次のように使います。

```
if(feof(some_stream))
    /* We're at the end */
```

clearerr関数は、ストリーム**stream**の**EOF**インジケータとエラーインジケータをクリアします。戻り値はなく、定義されているエラーもありません。この関数は、ストリームのエラー状態を解除するときに使います。

3.3.4 ストリームとファイルデスクリプタ

各ファイルストリームは、低水準ファイルデスクリプタに関連付けられています。低水準入出力操作と高水準ストリーム操作とを混在させることも可能ですが、バッファリングによる影響を予測することが困難なため、賢明な方法とはいえません。

```
#include <stdio.h>

int fileno(FILE *stream);
FILE *fdopen(int fildes, const char *mode);
```

fileno関数を呼び出すと、ファイルストリームが使っている低水準ファイルデスクリプタを調べることができます。成功すると、指定されたストリームのファイルデスクリプタを返し、失敗した場合には-1を返します。

fdopen関数を呼び出すと、すでにオープンされているファイルデスクリプタを使って新しいファイルストリームを作成することができます。この関数は、すでにオープンされているファイルデスクリプタに標準入出力ライブラリのバッファを提供する目的で使います。

fdopen関数は**fopen**関数と同じような働きをしますが、ファイル名の代わりに低水準ファイルデスクリプタを指定します。パラメータ**mode**の指定方法は、**fopen**と同じです。この指定は、ファイルが最初にオープンされたときに指定されたアクセスモードと矛盾してはいけません。**fdopen**は、成功すると新しいファイルストリームを返し、失敗すると**NULL**を返します。

3.4 ファイルとディレクトリの保守

標準ライブラリとシステムコールを利用すれば、ファイルとディレクトリの作成、保守に必要なすべての作業を行うことができます。

◆ chmod

chmod システムコールを使うと、ファイルやディレクトリのパーミッションを変更することができます。**chmod** システムコールは、**chmod** プログラムでも使われています。

```
#include <sys/stat.h>

int chmod(const char *path, mode_t mode);
```

chmodは、**path**で指定されたファイルのパーミッションを**mode**に変更します。モードには**open**システムコールの場合と同様、必要なパーミッションのビット単位の論理和を指定します。プログラムに適切な権限が与えられている場合を除けば、ファイルのパーミッションを変更できるのは、ファイルのオーナーかスーパーユーザーだけです。

◆ chown

スーパーユーザーは、**chown** システムコールを使ってファイルのオーナーを変更することができます。

```
#include <unistd.h>

int chown(const char *path, uid_t owner, gid_t group);
```

このシステムコールでは、ユーザーIDとグループIDの数値(**getuid**と**getgid**で得られる値)、およびだれがファイルのオーナーを変更できるかを制限する定数(**_POSIX_CHOWN_RESTRICTED**)を使います。適切な権限が設定されている場合には、ファイルのオーナーとグループが変更されます。

メモ

POSIXでは、実際にはスーパーユーザー以外のユーザーがファイルのオーナーを変更できるシステムを許容しています。現実のPOSIXシステムは、いずれもこのような操作を許していませんが、厳密にいえばこれは拡張仕様(FIPS 151-2)になります。本書が対象としているシステムは、いずれもXSI準拠であり、オーナー変更の制限は有効になっています。

◆ unlink、link、symlink

unlinkはファイルを削除します。

```
#include <unistd.h>

int unlink(const char *path);
int link(const char *path1, const char *path2);
int symlink(const char *path1, const char *path2);
```

unlinkシステムコールは、ファイルのリンクカウントをデクリメントします。成功すると0を返し、エラーが発生した場合には-1を返します。このシステムコールを使う場合には、該当するファイルのディレクトリエントリを持つディレクトリに対して、書き込みと実行のパーミッションを持っていなければなりません。

リンクカウントが0になり、どのプロセスもファイルをオープンしていなければ、ファイルは削除されます。実際には、ディレクトリエントリが削除されたあとも、最後のプロセスがファイルをクローズするまで、ファイルが使用している領域は解放されません。**rm**プログラムも、この**unlink**システムコールを使っています。追加のリンクは、ファイルに別の名前が付けられていることを意味します。通常、このようなリンクは**ln**プログラムによって作成されます。プログラムの中からファイルへの新しいリンクを作成するには、**link**システムコールを使います。

linkシステムコールは、**path1**で指定された既存のファイルへの新しいリンクを作成します。このとき、**path2**には新しいディレクトリエントリを指定します。同様の方法で**symlink**システムコールを使えば、シンボリックリンクを作成することができます。

◆ mkdir、rmdir

ディレクトリを作成または削除するには、**mkdir**システムコールと**rmdir**システムコールを使います。

```
#include <sys/stat.h>

int mkdir(const char *path, mode_t mode);
```

ディレクトリを作成する**mkdir**システムコールは、**mkdir**プログラムに相当するシステムコールです。**mkdir**システムコールは、**path**で指定された名前を持つディレクトリを作成します。ディレクトリのパーミッションはパラメータ**mode**で指定します。指定の方法は、**open**システムコールの**O_CREAT**オプションの場合と同じで、**umask**が適用される点も変わりません。

```
#include <unistd.h>

int rmdir(const char *path);
```

rmdirシステムコールはディレクトリを削除します。ただし、削除が成功するのは、ディレク

トリが空の場合だけです。**rmdir**プログラムは、このシステムコールを使っています。

◆ chdir、getcwd

ユーザーがファイルシステムの中を移動する場合とほぼ同様に、プログラムでもディレクトリを移動することができます。シェルでは**cd**コマンドを使ってディレクトリを変更しますが、プログラムでは**chdir**システムコールを使います。

```
#include <unistd.h>

int chdir(const char *path);
```

また、**getcwd**ライブラリ関数を呼び出すと、現在の作業ディレクトリを取得することができます。

```
#include <unistd.h>

char *getcwd(char *buf, size_t size);
```

getcwd関数は、現在のディレクトリの名前を**buf**で指定されたバッファに書き込みます。ディレクトリ名が**size**に指定されたバッファのサイズを超える場合(**ERANGE**エラーの場合)には、**NULL**を返します。成功すると**buf**を返します。

メモ

getcwdは、プログラムの実行中にディレクトリが削除された場合(**EINVAL**)、またはパーミッションが変更された場合(**EACCESS**)にも**NULL**を返します。

3.5 ディレクトリの走査

UNIXシステムでよく問題になるのは、ディレクトリの走査方法、つまりある特定のディレクトリにどのようなファイルが存在するかを調べる方法です。シェルプログラムでは、シェルにワイルドカードを展開させればよいので簡単ですが、プログラムではそういうわけにいきません。これまで、UNIXシステムでは、プログラムの中から低水準ファイルシステム構造にアクセスするために、システムごとに異なる方法を用いてきました。もちろん、ディレクトリを通常のファイルとしてオープンし、直接ディレクトリエントリを読み取ることも可能ですが、システムによってファイルシステムの構造と実装方法が異なるので、この方法には移植性がありません。しかし、現在では一連の標準ライブラリ関数が開発されており、以前と比べてディレクトリの走査も比較的簡単に行えるようになっています。

ディレクトリ関連の関数は、ヘッダーファイル**dirent.h**で宣言されており、構造体**DIR**を使ってディレクトリを操作するようになっています。この構造体へのポインタは**ディレクトリストリーム**（**DIR ***）と呼ばれ、通常のファイル操作で使うファイルストリーム（**FILE ***）と同じような働きをします。**DIR**構造体は直接変更すべきでものではないので、ディレクトリエントリ自体は、**dirent.h**で宣言されている構造体**dirent**で返されるようになっています。

ここでは、次の関数を取り上げます。

- **opendir**、**closedir**
- **readdir**
- **telldir**
- **seekdir**

◆ opendir

opendir関数は、ディレクトリをオープンし、ディレクトリストリームを確立します。成功すると、ディレクトリエントリの読み取りに使用できる**DIR**構造体へのポインタを返します。

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *name);
```

opendirは、失敗するとヌルポインタを返します。ディレクトリストリームでは低水準ファイルデスクリプタを使うので、オープンされているファイルが多すぎると、**opendir**が失敗することがあります。

◆ readdir

```
#include <sys/types.h>
#include <dirent.h>

DIR *readdir(DIR *dirp);
```

readdir関数は、**dirp**で指定されたディレクトリストリーム中の次のディレクトリエントリを示す構造体へのポインタを返します。以後は、**readdir**が呼び出されるたびに、次のディレクトリエントリを返します。ディレクトリの終わりに達すると、**NULL**を返します。

ほかのプロセスが同じディレクトリでファイルを作成したり削除したりしている場合、**readdir**では、そのディレクトリ（とサブディレクトリ）のすべてのファイルを取得できないことがあります。

dirent構造体には、各ディレクトリエントリについての情報を持つ次の要素が含まれています。

- ino_t d_ino ファイルのiノード
- char d_name[] ファイルの名前

ディレクトリ内のファイルについてさらに詳しい情報が必要な場合には、**stat**を呼び出す必要があります。

◆ telldir

```
#include <sys/types.h>
#include <dirent.h>

long int telldir(DIR *dirp);
```

telldir関数は、ディレクトリストリーム中の現在の位置を記録している値を返します。この値を使って**seekdir**を呼び出せば、ディレクトリの走査を現在の位置に再設定することができます。

◆ seekdir

```
#include <sys/types.h>
#include <dirent.h>

void seekdir(DIR *dirp, long int loc)
```

seekdir関数は、**dirp**で指定されたディレクトリストリーム中のディレクトリエントリポインタを設定します。位置の指定に使う**loc**の値は、あらかじめ**telldir**を呼び出して取得しておく必要があります。

◆ closedir

```
#include <sys/types.h>
#include <dirent.h>

int closedir(DIR *dirp);
```

closedir関数は、ディレクトリストリームをクローズし、関連付けられていたリソースを解放します。成功すると0を返し、エラーが発生した場合には-1を返します。

例題 ディレクトリ走査プログラム

次に示すprintdir.cは、ディレクトリ関連の関数とすでに説明したファイル操作関数とを組み合わせ、ディレクトリのリストを表示するプログラムです。このプログラムでは、ディレクトリ内の各ファイルを1行に1つずつ表示します。サブディレクトリの名前には最後にスラッシュを付け、その中のファイルは4文字分インデントして表示します。

プログラムでは、各サブディレクトリに移動してその中のファイルを調べます。見つかったファイルがディレクトリの場合には、その名前を直接opendirに渡し、再帰的に動作するようにします。ただし、オープン可能なディレクトリストリームの数は限られているので、非常に深くネストしたディレクトリがある場合には、エラーが発生する可能性があります。

走査するディレクトリをコマンドラインの引数で受け取るようにすれば、もっと汎用的なプログラムにすることができます。その場合には、lsやfindといったユーティリティのソースコードを参照して、どのような処理を行っているか参考にするとよいでしょう。

1 まず必要なヘッダーファイルをインクルードしたあと、現在のディレクトリを表示するprintdir関数を記述します。この関数は、見つかったサブディレクトリに対して再帰的に実行されます。パラメータdepthは、インデントの文字数の指定です。

```
#include <unistd.h>
#include <stdio.h>
#include <dirent.h>
#include <string.h>
#include <sys/stat.h>

void printdir(char *dir, int depth)
{
  DIR *dp;
  struct dirent *entry;
  struct stat statbuf;

  if((dp = opendir(dir)) == NULL) {
    fprintf(stderr,"cannot open directory: %s\n", dir);
    return;
  }
  chdir(dir);
  while((entry = readdir(dp)) != NULL) {
    stat(entry->d_name,&statbuf);
    if(S_ISDIR(statbuf.st_mode)) {
      /* Found a directory, but ignore . and .. */
      if(strcmp(".",entry->d_name) == 0 ||
         strcmp("..",entry->d_name) == 0)
           continue;
      printf("%*s%s/\n",depth,"",entry->d_name);
      /* Recurse at a new indent level */
      printdir(entry->d_name,depth+4);
    }
    else printf("%*s%s\n",depth,"",entry->d_name);
```

```
    }
    chdir("..");
    closedir(dp);
}
```

2 次は`main`関数です。

```
int main()
{
    printf("Directory scan of /home/neil:\n");
    printdir("/home/neil",0);
    printf("done.\n");

    exit(0);
}
```

プログラムを実行すると、次のように出力されます。

```
$ printdir
Directory scan of /home/neil:
.less
.lessrc
.term/
    termrc
.elm/
    elmrc
Mail/
    received
    mbox
.bash_history
.fvwmrc
.tin/
    .mailidx/
    .index/
        563.1
        563.2
posted
attributes
active
tinrc
done.
```

解説

処理の大半は`printdir`関数が行うので、この関数を中心に説明します。まず、`opendir`を使って簡単なエラーチェックを行い、目的のディレクトリが存在しているかどうかを確認します。ディレクトリが存在する場合には、`chdir`を呼び出してそのディレクトリに移動します。`readdir`が返したエントリが`NULL`でなければ、そのエントリがディレクトリかどうかを調べま

す。ディレクトリでなければ、depthで指定されたインデントを行ってファイルのエントリを表示します。

エントリがディレクトリの場合には、再帰的に処理を実行します。.（現在のディレクトリ）と..（親ディレクトリ）をスキップしたあと、再度printdir関数を呼び出し、同じ処理を繰り返します。ループから抜けたら、chdir("..")でもとのディレクトリに戻り、1つ上のディレクトリで中断していた処理を再開します。最後のclosedir(dp)は、必要以上にディレクトリストリームをオープンしないためのステートメントです。

このプログラムでは、走査するディレクトリを/home/neilとしてハードコーディングしているので、汎用性がありません。しかし、次のようにmainを書き換えると、より汎用性のある有益なディレクトリ表示プログラムになります。

```
int main(int argc, char* argv[])
{
  char *topdir, pwd[2]=".";
  if (argc !=2)
    topdir=pwd;
  else
    topdir=argv[1];

  printf("Directory scan of %s\n", topdir);
  printdir(topdir, 0);
  printf("done.\n");

  exit(0);
}
```

わずか数行を編集、追加するだけで、ディレクトリ名を引数に取る汎用ユーティリティになりました。ディレクトリ名が指定されなかった場合には、現在のディレクトリが処理の対象になります。作成したプログラムは、ほかのコマンドと組み合わせて利用することができます。

```
$ printdir /usr/local | more
```

このコマンドを実行すると、出力が画面単位で表示されるので、必要に応じてディレクトリツリーの前後を参照することができます。簡単な修正を加えるだけで、小さいながらも便利なディレクトリツリーブラウザを作成できました。さらに工夫すれば、ディレクトリの使用状況を表示したり、表示するディレクトリのレベルを指定したりすることもできます。

3.6 エラー

この章で説明したシステムコールと関数は、なんらかの理由で失敗することがあります。失敗した場合には、すでに何度か説明したように、エラーの原因を示す値を外部変数errnoに設定します。多くのライブラリでは、問題を報告するためにこの変数を使っています。このため、すでに指摘したように、errnoの値は該当する関数を呼び出した直後に調べる必要があります。ほかの関数を呼び出したあとでは、たとえその関数でエラーが発生しなくても、errnoの値が書き換えられてしまう可能性があります。

エラーを示す値とその意味は、ヘッダーファイルerrno.hに記述されています。次に示すのは、そのうちの代表的なものです。

- **EPERM** 操作は許可されていない。
- **ENOENT** 該当するファイルまたはディレクトリが存在しない。
- **EINTR** システムコールが割り込まれた。
- **EIO** I/Oエラー。
- **EBUSY** デバイスまたはリソースが使用中。
- **EEXIST** ファイルが存在する。
- **EINVAL** 引数が無効。
- **EMFILE** オープンされているファイルが多すぎる。
- **ENODEV** 該当するデバイスが存在しない。
- **EISDIR** ディレクトリである。
- **ENOTDIR** ディレクトリではない。

strerror関数とperror関数は、発生したエラーを報告するために使用できる便利な関数です。

```
#include <string.h>

char *strerror(int errnum);
```

strerror関数は、指定されたエラー番号のエラーの種類を説明する文字列を返します。発生したエラーの記録をとるときに使うと便利です。

```
#include <stdio.h>

void perror(const char *s);
```

perror関数は、errnoが示す現在のエラーの種類を説明する文字列を標準エラー出力に出力

します。このとき、文字列sがNULLでなければ、まずsで指定されたメッセージを表示し、そのあとにコロンと空白を出力してから、エラーの種類を説明する文字列を出力します。たとえば、次のように使います。

```
perror("program");
```

次に示すのは、標準エラー出力への出力例です。

```
program: Too many open files
```

3.7 その他のシステムコール

ここで取り上げるシステムコールはあまり使われることはありませんが、ある特定の状況では便利なことがあります。ここでは、あとで必要になった場合のためにこの種の関数についても解説しておきます。

◆ fcntl

fcntlシステムコールは、低水準ファイルデスクリプタを操作するためのさまざまな方法を提供します。

```
#include <fcntl.h>

int fcntl(int fildes, int cmd);
int fcntl(int fildes, int cmd, long arg);
```

fcntlシステムコールを使うと、ファイルデスクリプタの複製、ファイルデスクリプタフラグの取得と設定、ファイルステータスフラグの取得と設定、アドバイザリファイルロックなど、オープンされているファイルデスクリプタに対してさまざまな操作を行うことができます。

実行する操作は、コマンドパラメータ**cmd**によって指定します。指定する値は、**fcntl.h**で定義されています。選択したコマンドによっては、3つ目のパラメータ**arg**が必要になることがあります。以下では、各操作の構文について簡単に説明します。

```
fcntl(fildes, F_DUPFD, newfd);
```

この呼び出しは、整数**newfd**と等しいか、またはそれより大きい値の新しいファイルデスクリプタを返します。新しいファイルデスクリプタは、ファイルデスクリプタ**fildes**のコピーです。オープンされているファイルの数と**newfd**の値によっては、この呼び出しは実質的に

dup(fildes)と同じになります。

```
fcntl(fildes, F_GETFD);
```

この呼び出しは、fcntl.hで定義されているファイルデスクリプタフラグ（通常はFD_CLOEXECのみ）を取得します。FD_CLOEXECは、exec族のシステムコールに対する呼び出しの成功後にファイルデスクリプタをクローズするかどうかを示すフラグです。

```
fcntl(fildes, F_SETFD, flags);
```

この呼び出しは、ファイルデスクリプタフラグ（通常はFD_CLOEXECのみ）を設定するのに使います。

```
fcntl(fildes, F_GETFL);
fcntl(fildes, F_SETFL, flags);
```

これらの呼び出しは、ファイルステータスフラグとアクセスモードを設定または取得します。ファイルのアクセスモードは、fcntl.hで定義されているマスクO_ACCMODEを使うことで取得できます。その他のフラグとしては、O_CREATを指定してopenを呼び出す際に3番目の引数で指定するフラグがあります。fcntlでは、すべてのフラグを設定できるわけではないことに注意が必要です。特に、ファイルのパーミッションは、fcntlで設定することはできません。

fcntlを使ってアドバイザリファイルロックを実装することもできます。詳細については、マニュアルページのセクション2を参照してください。本書では、第7章でファイルのロックについて詳しく取り上げます。

◆ mmap

UNIXには、複数のプログラムでメモリを共有するための便利な機能があります。この機能は、2.0以降のLinuxカーネルに取り入れられています。mmap（メモリマップの意）はこの機能を実現する関数で、複数のプログラムで読み書き可能なメモリ領域をセットアップします。この場合、いずれかのプログラムが加えた変更は、ほかのプログラムからも見えます。

この機能はファイルを操作する場合にも利用でき、ディスクファイルの内容全体をメモリ内の配列のように扱うことができます。ファイルが、Cの構造体で記述可能なレコードから構成されている場合には、構造体の配列にアクセスする方法を使ってファイルを更新することができます。

この機能は、特別なパーミッションが設定された仮想メモリセグメントを使うことで実現されています。このセグメントに対して読み書きを行うと、オペレーティングシステムはディスクファイルの対応する部分に対して読み書きを実行します。

mmap関数は、オープンされているファイルデスクリプタを介してアクセスされるファイルの内容に関連付けられたメモリ領域へのポインタを作成します。構文は次のとおりです。

```
#include <sys/mman.h>

void *mmap(void *addr, size_t len, int prot, int flags, int fildes, off_t off);
```

共有セグメントによってアクセスされるファイルデータの先頭位置は、パラメータ**off**で変更できます。オープンされているファイルデスクリプタはパラメータ**fildes**で、アクセスできるデータの量（メモリセグメントの長さ）はパラメータ**len**で指定します。

パラメータ**addr**は、特定のメモリアドレスを要求するために使用できます。**addr**が0ならば、返されるポインタは自動的に割り当てられます。

パラメータ**prot**は、メモリセグメントのアクセスパーミッションを設定するのに使用し、次の定数値のビット単位の論理和によって指定します。

- **PRO_READ**　　セグメントは読み取り可能
- **PRO_WRITE**　　セグメントは書き込み可能
- **PRO_EXEC**　　セグメントは実行可能
- **PRO_NONE**　　セグメントはアクセス不可

flagsは、プログラムからセグメントに対して加えられた変更の反映方法を指定するパラメータです。

- **MAP_PRIVATE**　　セグメントはプライベートであり、変更はローカルになる。
- **MAP_SHARED**　　セグメントに対する変更はファイルに対して行われる。
- **MAP_FIXED**　　セグメントは指定したアドレス**addr**になければばならない。

msync関数は、メモリセグメントの一部または全部に対する変更内容を、メモリにマップされているファイルに書き込み（または読み取り）ます。

```
#include <sys/mman.h>

int msync(void *addr, size_t len, int flags);
```

セグメントのうち更新すべき部分は、開始アドレス**addr**と長さ**len**で指定します。**flags**は、更新の方法を指定するパラメータです。

- **MS_ASYNC**　　非同期書き込みを行う。
- **MS_SYNC**　　同期書き込みを行う。
- **MS_INVALIDATE**　　ファイルからデータを読み直す。

munmap関数は、指定されたメモリセグメントを解放します。

```
#include <sys/mman.h>

int munmap(void *addr, size_t len);
```

次のプログラムmmap_eg.cでは、mmapと配列へのアクセスを使って、構造体から構成されているファイルを更新します。このプログラムは、SunOSやほかのシステムでも正しく動作します。

例題 mmapの使用

1 最初にRECORD構造体を定義し、各レコードの番号を収めたNRECORDS個のデータを作成します。次に、作成したレコードをrecords.datファイルに追加します。

```
#include <unistd.h>
#include <stdio.h>
#include <sys/mman.h>
#include <fcntl.h>

typedef struct {
    int integer;
    char string[12];
} RECORD;

#define NRECORDS (100)

int main()
{
    RECORD record, *mapped;
    int i, f;
    FILE *fp;

    fp = fopen("records.dat","w+");
    for(i=0; i<NRECORDS; i++) {
        record.integer = i;
        sprintf(record.string,"RECORD-%d",i);
        fwrite(&record,sizeof(record),1,fp);
    }
    fclose(fp);
```

2 レコード43の整数値を143に変更し、これに合わせて該当レコードの文字列も書き換えます。

```
    fp = fopen("records.dat","r+");
    fseek(fp,43*sizeof(record),SEEK_SET);
    fread(&record,sizeof(record),1,fp);
    record.integer = 143;
    sprintf(record.string,"RECORD-%d",record.integer);
```

```
    fseek(fp,43*sizeof(record),SEEK_SET);
    fwrite(&record,sizeof(record),1,fp);
    fclose(fp);
```

3 今度は、メモリマップを使用してレコードをメモリにマップし、レコード43にアクセスして整数値を243に書き換えます。また、該当レコードの文字列も書き換えます。

```
    f = open("records.dat",O_RDWR);
    read(f,&record,sizeof(record));
    mapped = (RECORD *)mmap(0, NRECORDS*sizeof(record),
                          PROT_READ | PROT_WRITE, MAP_SHARED, f, 0);
    mapped[43].integer = 243;
    sprintf(mapped[43].string,"RECORD-%d",mapped[43].integer);
    msync((void *)mapped, NRECORDS*sizeof(record), MS_ASYNC);
    munmap((void *)mapped, NRECORDS*sizeof(record));
    close(f);

    exit(0);
}
```

本書のあとの章では、もうひとつの共有メモリ機構であるSystem V共有メモリを取り上げます。

3.8 この章のまとめ

この章では、UNIXがファイルとデバイスへのダイレクトアクセスをどのような方法で実現しているかを説明しました。また、低水準関数を土台として構築されたライブラリ関数を取り上げ、これらの関数を使うことでプログラミング上の課題に柔軟に対処できることを示しました。例として用いたディレクトリ走査プログラムはわずか数十行のコードですが、十分に強力なプログラムになっています。

この章で学んだファイルとディレクトリの操作方法を利用すれば、第2章で作成した初期バージョンのCDデータベースアプリケーションをCで書き直し、より構造化されたファイルベースのプログラムにすることができます。この章ではCDデータベースアプリケーションに新しい機能を追加することはできませんでしたが、以降の章で画面とキーボードの制御方法について学んだあと、実際にCでプログラムを書き直します。

第4章

UNIX環境

UNIXでプログラムを作成する場合には、プログラムが**マルチタスク環境**で実行される点を考慮しなければなりません。プログラムがマルチタスク環境で実行されるということは、ほかにも同時に実行されるプログラムがあるということです。つまり、メモリやディスク、CPU時間を、同時に実行されているほかのプログラムと共有することになります。また、同じプログラムの複数のインスタンスが並行して実行される可能性もあります。したがって、これらのプログラムが互いに干渉しないように、実行環境を意識して適切に動作するプログラムを作成しなければなりません。

この章では、プログラムが実行される環境や、動作条件についての情報を環境から取得する方法、ユーザーによるプログラムの動作の変更方法などについて説明します。取り上げる項目は次のとおりです。

- プログラムへの引数の渡し方
- 環境変数
- 時刻の取得
- 一時ファイル
- ユーザーとホストコンピュータに関する情報の取得
- ログメッセージの出力と設定
- システムに依存する制限の取得

4.1 プログラムの引数

Cで記述されたUNIXプログラムを実行すると、`main`関数から実行が開始されます。UNIXプログラムの場合、`main`関数は次のように宣言されます。

```
int main(int argc, char *argv[])
```

`argc`はプログラムの引数の数で、`argv`は引数自体を表す文字列の配列です。プログラムによっては、`main`を次のように宣言しているものもあります。

```
main()
```

戻り値の型はデフォルトで`int`になり、関数内で使われていない仮引数は宣言する必要がないので、上のように記述してもプログラムは動作します。`argc`と`argv`は存在していますが、宣言しないかぎり使用することはできません。

オペレーティングシステムは、新しくプログラムを実行する際にパラメータ`argc`と`argv`を設

定し、これをmainに渡します。通常、これらのパラメータはほかのプログラムから渡されます。ほとんどの場合、この処理は、オペレーティングシステムに対して新しいプログラムの実行を要求したシェルによって行われます。シェルは、入力されたコマンドラインを受け取って単語に分割したあと、配列argvを作成します。DOSではワイルドカードで指定された引数をプログラム自体が受け取ることを想定していますが、UNIXでは、argcとargvが設定される前に、シェルによってファイル名引数のワイルドカードが展開されます。

たとえば、シェルで次のようなコマンドを入力したとします。


```
$ myprog left right 'and center'
```

この場合、プログラムmyprogは、次の引数が渡されたものとしてmain関数から実行を開始します。

```
argc: 4
argv: {"myprog", "left", "right", "and center"}
```

引数の個数にはプログラム自体の名前も含まれており、配列argvの最初の要素argv[0]には、プログラムの名前がセットされます。シェルのコマンドラインで最後に引用符で囲んで指定した文字列は、空白を含む文字列からなる引数として指定されています。

ISO/ANSI Cに準拠したプログラムを記述した経験があれば、これらのことはすでによく知っているでしょう。main関数の引数は、シェルスクリプトの定位置パラメータ$0、$1などに対応します。ISO/ANSI Cではmainの戻り値の型を特定していませんが（voidまたはintを使用可能）、X/Open仕様では、この章の最初に示した明示的な宣言を使うことになっています。

コマンドライン引数は、必要な情報をプログラムに渡すときに便利です。たとえば、データベースアプリケーションの場合、使用するデータベースの名前を引数として渡せるようにしておくと、1つのプログラムでさまざまなデータベースを処理することができます。実際、多くのユーティリティプログラムでは、コマンドライン引数を使って動作を変更したり、オプションを設定したりできるようになっています。これらのコマンドライン引数は、**フラグ**や**スイッチ**と呼ばれ、一般的にはコマンドライン引数の前にダッシュ（-）を付けて指定します。たとえば、sortプログラムには、ソート順を逆にするための-rスイッチがあります。

```
$ sort -r file
```

コマンドラインオプションを使うプログラムは非常に多く存在します。もしオプションの指定方法が統一されていれば、プログラムもそれだけ使いやすくなります。従来は、各ユーティリティがそれぞれ独自の指定方法を用いていたため、コマンドが異なるとオプションの指定方法も異なるのがふつうでした。次の例を見てください。

```
$ tar cvfB /tmp/file.tar 1024
$ dd if=/dev/fd0 of=/tmp/file.dd bs=18k
```

```
$ ls -lstr
$ ls -l -s -t -r
```

このように、オプションの指定方法はコマンドによってまちまちです。また、プログラムによっては、**+x**のように指定すると、**-x**とは逆のオプションが有効になるものもあります。

こうしたプログラムのオプションの順序と意味をすべて記憶することは非常に困難です。通常は、**-h**オプションを指定してヘルプを表示するか、マニュアルページを参照してオプションの指定方法を確認するしかありません。ただし、こうした問題をスマートに解決できる方法もあります。それは、**getopt**を使うことです。**getopt**についてはすぐあとで取り上げます。ここでは、プログラムに渡された引数を取り出すサンプルプログラムを作成してみましょう。

例題 プログラムの引数

次のプログラム**args.c**は、プログラムの引数を表示します。

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int arg;

  for(arg = 0; arg < argc; arg++) {
    if(argv[arg][0] == '-')
      printf("option: %s\n", argv[arg]+1);
    else
      printf("argument %d: %s\n", arg, argv[arg]);
  }
  exit(0);
}
```

このプログラムを実行すると、指定された引数とオプションが表示されます。このプログラムでは、1つの文字列引数と、**-f**オプションに続けて指定されたファイル名引数を受け取ることを想定しています。もちろん、ほかのオプションを定義することも可能です。

```
$ args -i -lr 'hi there' -f fred.c
argument 0: args
option: i
option: lr
argument 3: hi there
option: f
argument 5: fred.c
```

解説

引数の個数を表す**argc**を使って、プログラムのすべての引数を調べるループを用意していま

す。引数がオプションかどうかは、先頭がダッシュかどうかで判定します。

ただし、たとえば**-l**と**-r**をどちらもオプションとして使う場合には、**-lr**を**-l -r**と同じように扱う必要があります。

X/Open仕様では、コマンドラインオプションの標準的な使い方が定義されており（Unix Syntax Guidelines)、Cプログラムでコマンドラインスイッチを使うための標準プログラミングインタフェースとして**getopt**関数が用意されています。

一般的なガイドラインとしては、次のようにするとよいでしょう。まず、コマンドラインスイッチはいずれもダッシュで始め、単一の文字または数字から構成されるようにします。引数を取らないオプションは、1つのダッシュのあとにまとめて指定できるようにします（たとえば、**ls**コマンドの場合、前に示した**ls -lstr**と**ls -l -s -t -r**は、オプションの指定としてはどちらも同じなので、このガイドラインが守られています)。また、オプションが値を取る場合には、その値を独立した引数としてオプションのあとに指定できるようにします。前にあげた例でいえば、**dd**のケースではこの原則が守られておらず、**tar**ではオプションとその値が離れた位置にあります。

◆ getopt

Linuxでは、上のようなガイドラインを簡単に実現できる**getopt**を利用することができます。**getopt**は、独自の値を取るオプションと取らないオプションの両方をサポートしています。

```
#include <unistd.h>

int getopt(int argc, char *const argv[], const char *optstring);
extern char *optarg;
extern int optind, opterr, optopt;
```

getopt関数は、プログラムの**main**関数に渡されるパラメータ**argc**と**argv**、およびオプション指定文字列を引数に取ります。オプション指定文字列は、プログラムで使用するオプションと、各オプションが独自の値を取るかどうかを指定するものです。この**optstring**は、文字を並べただけの単純なリストで、各文字が1文字のオプションを表します。文字のあとにコロン(**:**)が付いている場合には、そのオプションが引数を取ることを示します。**bash**で使用できる**getopts**コマンドも、この**getopt**とよく似た機能を持っています。

たとえば、上に示したサンプルプログラム**args.c**のオプション処理は、**getopt**を使って次のように記述することができます。

```
getopt(argc, argv, "if:lr");
```

上のように指定すると、引数を取らない単純なオプション**-i**、**-l**、**-r**と、ファイル名を引数に取る**-f**オプションを使用できるようになります。

getoptの戻り値は、配列**argv**の中で見つかった次のオプション文字です。したがって、実際

のプログラムでは、getoptを繰り返し呼び出して各オプションを順に調べることになります。getoptの動作は次のとおりです。

- オプションが値を取る場合には、外部変数optargにその値が格納されます。
- 処理すべきオプションがもうない場合、getoptは-1を返します。
- getoptは、認識できないオプションを外部変数optoptに格納し、?を返します。
- 値を必要とするオプション（上の-fオプションなど）に値が指定されていなかった場合、getoptは:を返します。

外部変数optindには、処理すべき次の引数のインデックス（添字）がセットされます。getoptは、このoptindを使って、どこまで処理を行ったかを記憶しています。プログラムでこの変数を設定する必要がある場合はまれです。すべてのオプション引数の処理が終わったあとのoptindは、配列argvの中の残りの引数がある場所を指しています。

getoptのバージョンによっては、オプション以外の引数に出合った時点で-1を返し、これに合わせてoptindを設定するものがあります。その他のバージョン、たとえばLinuxで使用できるgetoptは、引数の中のどの位置にあるかにかかわらずオプションを処理します。ただし、この場合、argvが実際にはgetoptによって書き換えられ、オプション以外の引数は、argv[optind]から始まる要素としてまとめて渡されることに注意する必要があります。また、getoptの実装によっては、不明なオプションが指定された場合にエラーメッセージを表示するものもあります。POSIX仕様では、変数opterrに0以外の値が設定されている場合、getoptはstderrにエラーメッセージを表示することになっています。

例題 getopt

前に作成したプログラムをgetoptを使って書き換え、argopt.cを作成します。

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
  int opt;

  while((opt = getopt(argc, argv, "if:lr")) != -1) {
    switch(opt) {
    case 'i':
    case 'l':
    case 'r':
      printf("option: %c\n", opt);
      break;
    case 'f':
      printf("filename: %s\n", optarg);
```

```
        break;
      case ':':
        printf("option needs a value\n");
        break;
      case '?':
        printf("unknown option: %c\n", optopt);
        break;
      }
    }
    for(; optind < argc; optind++)
      printf("argument: %s\n", argv[optind]);
    exit(0);
}
```


このプログラムを実行すると、コマンドラインで指定した引数が自動的に処理されることがわかります。

```
$ argopt -i -lr 'hi there' -f fred.c -q
option: i
option: l
option: r
filename: fred.c
argopt: illegal option -- q
unknown option: q
argument: hi there
```

解説

このプログラムでは、**getopt**を繰り返し呼び出してオプション引数を処理しています。ループは、**getopt**が-1を返したとき、つまりオプション引数がなくなった時点で終了します。各オプションについては、不明なオプションや値が指定されていない場合も含めて、適切な処理を行います。

オプションをすべて処理し終わったら、**optind**から始まる残りの引数を表示します。

4.2 環境変数

環境変数については、すでに第2章で取り上げました。環境変数を使うと、シェルスクリプトやほかのプログラムの動作を制御することができます。環境変数は、ユーザーの環境を設定するために使用することもできます。たとえば、環境変数**HOME**はユーザーごとに設定され、この変数によって、ユーザーのセッションを開始するデフォルトの場所であるホームディレクトリが定義されます。すでに学んだように、環境変数はシェルのプロンプトから内容を調べることができます。

```
$ echo $HOME
/home/neil
```

UNIXでは多くの標準的な環境変数が定義されており、端末の種類やデフォルトエディタの指定、タイムゾーンの選択など、さまざまな目的のために使われています。Cプログラムでは、**putenv**関数と**getenv**関数を使うことで環境変数にアクセスできます。

```
#include <stdlib.h>

char *getenv(const char *name);
int putenv(const char *string);
```

環境は、**name=value**という形式の文字列から構成されています。**getenv**関数は、指定された名前を持つ文字列を環境から探し、その名前に関連付けられている値を返します。指定された変数が存在しない場合には、**NULL**を返します。変数が存在していても、値を持っていない場合、**getenv**の呼び出しは成功し、先頭バイトが**\0**の文字列が返されます。**getenv**が返す文字列は静的な記憶領域に格納されるので、アプリケーションで上書きしてはいけません。

putenv関数は、**name=value**という形式の文字列を引数に取り、これを現在の環境に追加します。新しい環境を割り当てるのに必要なメモリが不足している場合など、エラーが発生すると-1を返します。このエラーが返された場合、エラー変数**errno**には**ENOMEM**がセットされます。

では、これらの関数を使って、指定した環境変数の値を表示するプログラム**environ.c**を作成してみましょう。引数を2つ指定した場合には、環境変数の値も設定できるようにします。

例題 getenvとputenv

1 **main**関数の最初で、プログラムが正しく呼び出されたかどうかチェックします。

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
  char *var, *value;

  if(argc == 1 || argc > 3) {
    fprintf(stderr,"usage: environ var [value]\n");
    exit(1);
  }
```

2 次に、**getenv**を使って、指定された変数の値を環境から取得します。

```
  var = argv[1];
  value = getenv(var);
```

```
  if(value)
    printf("Variable %s has value %s\n", var, value);
  else
    printf("Variable %s has no value\n", var);
```

3 引数が2つ指定されたかどうかを調べます。2番目の引数が指定されている場合には、name=value形式の文字列を作成してputenvを呼び出し、1番目の引数で指定された変数に、2番目の引数で指定された値を設定します。


```
  if(argc == 3) {
    char *string;
    value = argv[2];
    string = malloc(strlen(var)+strlen(value)+2);
    if(!string) {
      fprintf(stderr,"out of memory\n");
      exit(1);
    }
    strcpy(string,var);
    strcat(string,"=");
    strcat(string,value);
    printf("Calling putenv with: %s\n",string);
    if(putenv(string) != 0) {
      fprintf(stderr,"putenv failed\n");
      free(string);
      exit(1);
    }
```

4 もう一度getenvを呼び出し、変数に設定された新しい値を確認します。

```
    value = getenv(var);
    if(value)
      printf("New value of %s is %s\n", var, value);
    else
      printf("New value of %s is null??\n", var);
  }
exit(0);
}
```

プログラムを実行すると、次のように出力されます。

```
$ environ HOME
Variable HOME has value /home/neil
$ environ FRED
Variable FRED has no value
$ environ FRED hello
Variable FRED has no value
Calling putenv with: FRED=hello
New value of FRED is hello
$ environ FRED
Variable FRED has no value
```

注意が必要なのは、環境がプログラムにローカルであるという点です。プログラムの中で変更を行っても、その変更はプログラムの外部には反映されません。これは、変数の値が子プロセス（実行したプログラム）から親プロセス（シェル）には伝えられないためです。

4.2.1 環境変数の利用

プログラムでは、しばしば環境変数を利用して動作を変更します。プログラムで使われる環境変数をユーザーが変更するには、ログインシェルによって読み取られるファイル**.profile**を使ってデフォルトの環境で変数に値を設定する方法と、シェルのコマンドラインで変数を指定する方法があります。次の例を見てください。

```
$ environ FRED
Variable FRED has no value
$ FRED=hello environ FRED
Variable FRED has value hello
```

シェルは、コマンドラインの最初に指定された変数の割り当てを環境変数に対する一時的な変更として受け取ります。したがって、上の2番目の例では、プログラム**environ**は、変数**FRED**が値を持つ環境で実行されることになります。

このことを利用すると、CDデータベースアプリケーションで使用するデータベースを**CDDB**などの環境変数で指定できるようにすることも可能です。こうすれば、各ユーザーは自分のデフォルトの設定を指定したり、アプリケーションを実行するたびに異なるデータベースを指定したりすることができます。たとえば、次のようにします。

```
$ CDDB=mycds; export CDDB
$ cdapp
```

一時的に環境を変更する場合には、次のようにしてアプリケーションを起動します。

```
$ CDDB=mycds cdapp
```

注意

環境変数には利点もあれば欠点もあるので、慎重に使用する必要があります。たとえば、環境変数は、コマンドラインオプションと比べてユーザーにとって見えにくい側面があり、それだけデバッグも難しくなります。プログラムの動作が変更されて予期しない結果を生じることがあるという点では、環境変数はグローバル変数に通じるものを持っているといえるでしょう。

4.2.2 変数environ

すでに説明したように、プログラムの環境はname=valueという形式の文字列から構成されています。この文字列配列は、変数environを介して直接プログラムから利用することができます。変数environは次のように宣言されています。

```
#include <stdlib.h>

extern char **environ;
```


例題 environ

次のプログラムshowenv.cは、変数environを使って環境変数を表示します。

```
#include <stdlib.h>
#include <stdio.h>

int main()
{
  char **env = environ;

  while(*env) {
    printf("%s\n",*env);
    env++;
  }
  exit(0);
}
```

Linux上でこのプログラムを実行すると、次のように表示されます（一部出力を省略しています）。

```
$ showenv
HOSTNAME=euripide.provider.com
LOGNAME=neil
MAIL=/var/spool/mail/neil
TERM=linux
HOSTTYPE=i386
PATH=/usr/local/bin:/bin:/usr/bin
HOME=/home/neil
LS_OPTIONS=--8bit --color=tty -F -b -T 0 -N
SHELL=/bin/bash
PS1=\h:\w\$
PS2=>
OSTYPE=Linux
```

解説

ヌルで終わる文字列配列である変数**environ**とループを使って、環境をすべて出力しています。

4.3 日付と時刻

プログラムがいつ実行されたかを記録する場合や、時刻によってプログラムの動作を変更する場合など、プログラムの中で日付と時刻を取得できると便利なことがあります。たとえば、勤務時間中には実行できないゲームを作成したり、深夜になるまで待機して自動的にバックアップを開始するプログラムを作成する場合などは、日付と時刻の値が役に立ちます。

メモ

すべてのUNIXシステムは、グリニッジ標準時（GMT）の1970年1月1日午前0時を日付と時刻の起点としています。これがいわゆる"エポックの起点"です。UNIXシステムで使われるすべての時間は、この時刻を起点に計算されます。MS-DOSでも同じような方法で時間を計算しますが、MS-DOSの場合には1980年からエポックが始まります。その他のシステムでは、独自のエポックの起点が使われています。

時刻は**time_t**型を使って処理されます。**time_t**型は、秒で表現された日付と時刻を格納できるだけの十分な大きさを持った整数型です。Linuxシステムでは**time_t**型は**long**で、時刻を操作する関数とともに、ヘッダーファイル**time.h**で定義されています。

```
#include <time.h>

time_t time(time_t *tloc);
```

低水準の時刻の値を取得するには、**time**関数を呼び出します。**time**関数は、エポックの起点からの経過秒数を返します。このとき、**tloc**がヌルポインタでなければ、**tloc**で示される場所にも経過秒数を書き込みます。

例題 time

次のプログラム**envtime.c**は、**time**関数の使い方の例です。

```
#include <time.h>
#include <stdio.h>
#include <unistd.h>

int main()
```

```
{
  int i;
  time_t the_time;

  for(i = 1; i <= 10; i++) {
    the_time = time((time_t *)0);
    printf("The time is %ld\n", the_time);
    sleep(2);
  }
  exit(0);
}
```


このプログラムを実行すると、2秒おきに低水準の時刻の値が表示されます。

```
$ envtime
The time is 910099647
The time is 910099649
The time is 910099651
The time is 910099653
The time is 910099655
The time is 910099657
The time is 910099659
The time is 910099661
The time is 910099663
The time is 910099665
```

解説

このプログラムでは、ヌルポインタを指定して**time**関数を呼び出しています。取得した時刻を表示して2秒間スリープしたあと、再び同じ処理を繰り返します。**time**関数を10回呼び出すと、プログラムは終了します。

1970年を起点とする秒数で表された日付と時刻は、なんらかの処理にかかった時間を計測するときに便利です。処理の始まりと終わりで**time**関数を呼び出し、返された2つの値に対して引き算を行うだけでよいからです。ただし、ISO/ANSI Cでは、秒単位での時間の計測には慎重を期して**time_t**型を指定せず、**difftime**関数を用意しています。**difftime**関数は、2つの**time_t**型の値の秒数の差を計算し、その結果を**double**として返します。

```
#include <time.h>

double difftime(time_t time1, time_t time2);
```

difftime関数は、2つの時刻の値の差を計算し、**time1-time2**の値を浮動小数点数として返します。UNIXの場合、**time**関数からの戻り値は秒数なので、これをもとに2つの時刻の差を計算することもできますが、移植性を考慮するなら**difftime**関数を使うべきです。

日付と時刻を人間にとってわかりやすい形で表現するには、低水準の時刻の値をしかるべき日

付と時刻に変換する必要があります。そのために用意されているのがgmtime関数です。

gmtime関数は、低水準の時刻の値を日常使う形式のフィールドからなる構造体に格納します。

```
#include <time.h>

struct tm *gmtime(const time_t timeval);
```

構造体tmは、少なくとも次のメンバを含むものとして定義されています。

表4.1 tm構造体のメンバ

メンバ	説明
int tim_sec	秒(0～61)
int tim_min	分(0～59)
int tim_hour	時間(0～23)
int tim_mday	日(1～31)
int tim_mon	月(0～11。0は1月)
int tim_year	年(1900年が起点)
int tim_wday	曜日(0～6。0は月曜日)
int tim_yday	1年の中での通算日数(0～365)
int tim_isdst	夏時間

tm_secが61までの値を取るのは、うるう秒を考慮しているためです。

例題 gmtime

次のプログラムgmtime.cは、構造体tmとgmtime関数を使って現在の日付と時刻を表示します。

```
#include <time.h>
#include <stdio.h>

int main()
{
  struct tm *tm_ptr;
  time_t the_time;

  (void) time(&the_time);
  tm_ptr = gmtime(&the_time);

  printf("Raw time is %ld\n", the_time);
  printf("gmtime gives:\n");
  printf("date: %02d/%02d/%02d\n",
         tm_ptr->tm_year, tm_ptr->tm_mon+1, tm_ptr->tm_mday);
  printf("time: %02d:%02d:%02d\n",
         tm_ptr->tm_hour, tm_ptr->tm_min, tm_ptr->tm_sec);
```

```
  exit(0);
}
```

このプログラムとdateコマンドを続けて実行すると、次のように表示されます。

```
$ gmtime; date
Raw time is 910102265
gmtime gives:
date: 98/11/03
time: 14:11:05
Tue Nov  3 14:11:05 GMT 1998
```


解説

このプログラムでは、time関数を呼び出して低水準の時刻の値を取得し、次にgmtimeを呼び出して日常使う日付と時刻の表現に変換しています。それぞれの値は、printfを使って出力しています。厳密にいえば、低水準の時刻の値を上のような方法で出力するのは避けたほうがよいでしょう。システムによっては、time_tがlongでない場合もあるからです。実行例では、gmtimeを実行した直後にdateコマンドを実行し、出力を比較しています。

このプログラムにはひとつ問題があります。それは、GMT以外のタイムゾーンや、夏時間が有効になっている場所でこのプログラムを実行すると、正しい時刻（および、場合によっては正しい日付）が表示されないことです。原因は、gmtimeがGMTの時間を返すためです（現在では、GMTはUTC、"協定世界時"と呼ばれています）。UNIXではこの仕様により、全世界を通じてすべてのプログラムとシステムが同期化されています。タイムゾーンが異なっていても、同じ時刻に作成されたファイルは、すべて同じ作成時刻を持つことになります。ローカルタイム、つまり地域ごとの時刻を取得するには、localtime関数を使います。

```
#include <time.h>

struct tm *localtime(const time_t *timeval);
```

localtime関数は、ローカルタイムゾーンと夏時間に関する調整を加えた値を返す点を除いて、gmtime関数と同じです。上のプログラムでgmtimeの代わりにlocaltimeを使えば、プログラムを実行した地域に応じた日付と時刻が返されます。

ところで、tm構造体に格納した値から逆にtime_t型の値を求めるには、どうすればよいでしょうか。この場合には、mktime関数を使います。

```
#include <time.h>

time_t mktime(struct tm *timeptr);
```

構造体をtime_tの値として表現できない場合、mktimeは-1を返します。

dateプログラムで表示されるようなわかりやすい形式で日付と時刻を表示したい場合には、asctime関数とctime関数を使います。

```
#include <time.h>

char *asctime(const struct tm *timeptr);
char *ctime(const time_t *timeval);
```

asctime関数は、tm構造体timeptrで指定された日付と時刻を表す文字列を返します。返される文字列は、次のような形式をしています。

```
Wed Nov  4 07:20:57 1998\n\0
```

この文字列は、26文字の長さを持つ固定形式の文字列です。ctime関数は、次の呼び出しと同じです。

```
asctime(localtime(timeval))
```

例題 ctime

次の例は、ctimeの動作を確認するプログラムです。

```
#include <time.h>
#include <stdio.h>

int main()
{
  time_t timeval;

  (void)time(&timeval);
  printf("The date is: %s", ctime(&timeval));
  exit(0);
}
```

プログラムを実行すると、次のように出力されます。

```
$ ctime
The date is: Wed Nov  4 16:26:42 1998
```

解説

このプログラムでは、time関数を呼び出して低水準の時刻の値を取得し、ctime関数を使ってわかりやすい文字列に変換したあと、結果を表示しています。

日付と時刻を表す文字列の書式を細かく設定したい場合には、最近のUNIXシステムに用意されている**strftime**関数を使用することができます。日付と時刻用の**sprintf**と考えるとよいでしょう。

```
#include <time.h>

size_t strftime(char *s, size_t maxsize, const char *format, struct tm
*timeptr);
```


strftime関数は、**timeptr**で指定された構造体**tm**が表す日付と時刻に書式を設定し、結果を文字列**s**に書き込みます。文字列**s**は、少なくとも**maxsize**文字の長さを持つものとして扱われます。文字列**format**は、文字列**s**に書き込む文字を指定するのに使います。**format**には、そのまま出力される通常の文字と、日付と時刻の要素に書式を設定するための変換指定子を指定できます。変換指定子には次のものがあります。

表4.2 strftime関数の変換指定子

変換指定子	説明
%a	曜日名の省略形
%A	省略しない曜日名
%b	月名の省略形
%B	省略しない月名
%c	日付と時刻
%d	日（01～31）
%H	24時間制の時間（00～23）
%I	12時間制の時間（01～12）
%j	1年の中での通算日数（001～366）
%m	月（01～12）
%M	分（00～59）
%p	AMまたはPM
%S	秒（00～61）
%U	1年の中での通算週数（01～53。日曜日を週の第1日とする）
%V	1年の中での通算週数（01～53。月曜日を週の第1日とする）
%w	曜日の番号（0～6。0は日曜日）
%x	地域の表示形式での日付
%X	地域の表示形式での時刻
%y	西暦年の下2桁（00～99）
%Y	西暦年
%z	タイムゾーン名
%%	%文字

たとえば、**date**プログラムが出力する日付の形式を**strftime**の書式指定文字列を使って表すと、次のようになります。

```
"%a %b %d %H:%M:%S %Y"
```

日付を読み取るときには、**strptime**関数を使うと便利です。**strptime**関数は、日付と時刻を表す文字列を受け取って**tm**構造体を作成します。

```
#include <time.h>

char *strptime(const char *buf, const char *format, struct tm
*timeptr);
```

文字列**format**は、**strftime**の場合と同様に指定します。**strptime**は、指定された文字列を走査してフィールドが見つかったら変数に格納する点では、**sscanf**に似ています。ただし、**strptime**関数では、文字列**format**に従って**tm**構造体のメンバに値が格納されます。**strptime**の変換指定子は、**strftime**の場合より制限がゆるくなっており、曜日と月の名前では省略した形と省略していない形のどちらも受け付けます。たとえば、"Tue" も "Tuesday" も**%a**変換指定子にマッチします。また、**strftime**では、10未満の数字の前には常に0が付きますが、**strptime**では先行する0を省略できます。

strptimeは、変換処理が行われた最後の文字の次の文字へのポインタを返します。変換できない文字に遭遇した場合には、その時点で変換が中止されます。このため、**strptime**を呼び出す場合には、渡された文字列のうち、必要な部分が処理されたかどうかを調べ、**tm**構造体に適切な値が書き込まれていることを確認する必要があります。

例題 strftimeとstrptime

次のプログラム**strftime.c**では、さまざまな変換指定子を使っています。前に示した表を参照しながら、それぞれどのように解釈されるか確かめてください。

```
#include <time.h>
#include <stdio.h>

int main()
{
  struct tm *tm_ptr, timestruct;
  time_t the_time;
  char buf[256];
  char *result;

  (void) time(&the_time);
  tm_ptr = localtime(&the_time);
  strftime(buf, 256, "%A %d %B, %I:%S %p", tm_ptr);

  printf("strftime gives: %s\n", buf);

  strcpy(buf,"Tue 4 December 1995, 17:53 will do fine");
```

```
    printf("calling strptime with: %s\n", buf);
    tm_ptr = &timestruct;

    result = strptime(buf,"%a %d %b %Y, %R", tm_ptr);
    printf("strptime consumed up to: %s\n", result);

    printf("strptime gives:\n");
    printf("date: %02d/%02d/%02d\n",
           tm_ptr->tm_year, tm_ptr->tm_mon+1, tm_ptr->tm_mday);
    printf("time: %02d:%02d\n",
           tm_ptr->tm_hour, tm_ptr->tm_min);
    exit(0);
}
```

4

プログラムを実行すると、次のように出力されます。

```
$ strftime
strftime gives: Wednesday 04 November, 07:02 PM
calling strptime with: Tue 4 December 1995, 17:53 will do fine
strptime consumed up to:  will do fine
strptime gives:
date: 95/12/04
time: 17:53
```

解説

まず、time関数とlocaltime関数を呼び出して現在の時刻（ローカルタイム）を取得します。次に、適切な書式指定文字列を指定してstrftime関数を呼び出し、わかりやすい形式に変換します。strptimeの使い方を示すために、日付と時刻を含む文字列を設定し、strptimeを呼び出して対応する値に変換し、その結果を出力しました。変換指定子%Rは、%H:%Mを省略した指定方法で、strptimeで使用することができます。

日付と時刻を正しく読み取るには、strptimeに正確な書式指定文字列を指定する必要があることに注意してください。書式が非常に限定されたものである場合以外は、ユーザーが入力した日付を正確に変換できるとは考えないほうがよいでしょう。

4.4 一時ファイル

プログラムでは、しばしばファイルによる一時的な記憶領域が必要になります。たとえば、計算の途中経過を記録したり、重要な操作を行う前にファイルのバックアップを作成したりすることがあります。特にデータベースアプリケーションでは、レコードを削除するときに**一時ファイル**を使うと便利です。削除せずに残すレコードを一時ファイルに書き込み、処理を終えたら一時

ファイルを新しいデータベースとして残し、それまで使っていたファイルを削除すればよいからです。

一時ファイルはよく使われますが、一時ファイルに使うファイル名が一意でなければならない点に注意する必要があります。ファイル名が一意でない場合、UNIXのようなマルチタスクシステムでは、同じファイル名を使うほかのプログラムとの間で衝突が起きることがあります。

一意のファイル名を生成するには、**tmpnam**関数を使います。

```
#include <stdio.h>

char *tmpnam(char *s);
```

tmpnam関数は、既存のどのファイルとも異なるファイル名を返します。文字列**s**がヌルでなければ、**s**で指定された文字列にもファイル名を書き込みます。**tmpnam**の戻り値のために用意されている静的な記憶領域は、**tmpnam**がさらに呼び出されたときに上書きされるので、**tmpnam**が何度も呼び出される場合には、引数の文字列を指定する必要があります。この文字列は、少なくとも**L_tmpnam**文字の長さを持つものとして扱われます。**tmpnam**は、1つのプログラムの中で**TMP_MAX**回まで呼び出すことができます。この場合、呼び出すたびに異なるファイル名が生成されます。

一時ファイルをすぐに使う場合には、**tmpfile**関数を使えば、一時ファイル名の作成とオープンを同時に行うことができます。**tmpnam**関数を使った場合、ほかのプログラムが**tmpnam**の返す名前と同じ名前でファイルを作成する可能性がありますが、**tmpfile**関数ではこの問題を回避することができます。

```
#include <stdio.h>

FILE *tmpfile(void);
```

tmpfile関数は、一意の一時ファイルへのストリームポインタを返します。ファイルは読み書き用にオープンされ（**w+**を指定して**fopen**が呼び出される）、ファイルへのすべての参照がクローズされた時点で自動的に削除されます。

エラーが発生した場合、**tmpfile**は**NULL**を返し、**errno**にエラーの内容を示す値を設定します。

例題 tmpnamとtmpfile

次に示すのは、**tmpnam**と**tmpfile**を使ったプログラム**tmpnam.c**です。

```
#include <stdio.h>

int main()
{
```

```
    char tmpname[L_tmpnam];
    char *filename;
    FILE *tmpfp;

    filename = tmpnam(tmpname);

    printf("Temporary file name is: %s\n", filename);

    tmpfp = tmpfile();
    if(tmpfp)
      printf("Opened a temporary file OK\n");
    else
      perror("tmpfile");
    exit(0);
}
```


プログラムを実行すると、**tmpnam**によって生成された一意のファイル名が表示されます。

```
$ tmpnam
Temporary file name is: /tmp/31056aaa
Opened a temporary file OK
```

解説

tmpnamを呼び出し、一時ファイルに使用する一意のファイル名を生成しています。実際に一時ファイルを使う場合には、すばやくファイルをオープンする必要があります。ファイル名を取得してからファイルをオープンするまで間をあけると、ほかのプログラムが同じ名前でファイルをオープンしてしまう可能性があるからです。このようなリスクは、**tmpfile**関数を使うことで回避できます。**tmpfile**関数では、ファイル名の作成と一時ファイルのオープンが同時に行われます。

古いバージョンのUNIXでは、別の方法として、**mktemp**関数と**mkstemp**関数を使って一時ファイル名を作成することができます。これらの関数は**tmpnam**と似ていますが、一時ファイル名のテンプレートを指定できる点が異なります。テンプレートでは、ファイルの場所と名前をある程度まで指定することが可能です。

```
#include <stdlib.h>

char *mktemp(char *template);
int mkstemp(char *template);
```

mktemp関数は、指定された**template**から一意のファイル名を作成します。**template**は、末尾が6文字の**X**からなる文字列でなければなりません。6文字の**X**の部分は、**mktemp**関数によって、ファイル名に使用できる文字の一意の組み合わせに置き換えられます。**mktemp**関数は、生成された文字列へのポインタを返します。一意のファイル名を作成できなかった場合には**NULL**

を返します。

mkstemp関数は、**tmpfile**関数と同様、一時ファイル名を作成してオープンします。ファイル名は**mktemp**と同じ方法で作成されますが、**mkstemp**が返すのは、オープンされた低水準ファイルデスクリプタです。一般には、**mktemp**と**mkstemp**ではなく、**tmpnam**と**tmpfile**を使ったほうがよいでしょう。

4.5 ユーザー情報

すべてのUNIXプログラムは、**init**を除いて、ほかのプログラムまたはユーザーから実行されます。通常、ユーザーはシェルからプログラムを実行します。このとき、すでに説明した方法を使って環境変数を調べたりシステムクロックを読み出したりすれば、プログラムが実行されている環境について知ることができます。ここでは、プログラムを使っているユーザーに関する情報を調べる方法について説明します。

ユーザーは、UNIXシステムにログインするときにユーザー名とパスワードを入力します。入力した内容が正しければ、ユーザーはシェルを使用できるようになります。内部的には、ユーザーは一意のユーザーID、いわゆる**UID**を持っています。UNIXで実行される各プログラムは、ユーザーになり代わる形で実行され、該当するUIDに関連付けられます。

プログラムに**setUID**パーミッションをセットすれば、あたかも別のユーザーが起動したかのようにプログラムを実行することができます。たとえば、**su**コマンドを実行すると、rootユーザーがプログラムを起動したかのように**su**コマンドが実行されます。**su**コマンドは、ユーザーのアクセスが有効かどうかを確認したあと、UIDをターゲットアカウントのものに変更し、このアカウントのログインシェルを実行します。このしくみを使うと、別のユーザーから起動されたものとしてプログラムを実行することができます。システム管理者は、しばしばこの方法を使ってシステムの保守作業を行います。

UIDはユーザーを識別する重要な情報なので、まずUIDから説明しましょう。

UIDには**uid_t**という独自の型があり、これは**sys/types.h**で定義されています。通常、UIDは小さな整数です。一部のUIDはシステムによってあらかじめ定義されており、ほかのUIDは、新しいユーザーをシステムに追加するときにシステム管理者が作成します。一般に、通常のユーザーのUIDは100より大きい値を持っています。

```
#include <sys/types.h>
#include <unistd.h>

uid_t getuid(void);
char *getlogin(void);
```

getuid関数は、プログラムに関連付けられているUIDを返します。これは、通常、プログラ

ムを起動したユーザーのUIDです。

getlogin関数は、現在のユーザーのログイン名を返します。

システムファイルの/etc/passwdには、ユーザーアカウントに関するデータベースが格納されています。ファイルは1行1ユーザーの形式で、各行にはユーザー名、暗号化されたパスワード、ユーザーID (UID)、グループID (GID)、フルネーム、ホームディレクトリ、デフォルトシェルが記述されています。

プログラムを起動したユーザーのUIDをプログラムの中で調べる場合、パスワードファイルを直接調べてユーザーのログイン名とフルネームを取得することもできます。しかし、この方法は勧められません。新しいUNIXシステムでは、セキュリティ上の理由から単純なパスワードファイルの使用を避ける傾向があり、暗号化されたパスワード情報を含まない"シャドウ"パスワードファイルを用いることが多いからです。現在では、ユーザー情報を効率的に取得するための標準的なプログラミングインタフェースとして、いくつかの関数が定義されています。

```
#include <sys/types.h>
#include <pwd.h>

struct passwd *getpwuid(uid_t uid);
struct passwd *getpwnam(const char *name);
```

pwd.hで定義されているパスワードデータベース構造体passwdは、次のメンバを含んでいます。

表4.3 パスワードデータベース構造体passwdのメンバ

メンバ	説明
char *pw_name	ユーザーのログイン名
uid_t pw_uid	UID
gid_t pw_gid	GID
char *pw_dir	ユーザーのホームディレクトリ
char *pw_shell	ユーザーのデフォルトシェル

UNIXシステムの中には、ユーザーのフルネーム用のフィールドを持つものがあります。ただし、このフィールドはシステムによってpw_gecosとなっていたり、pw_commentとなっていたりするなど、標準仕様がありません。したがって、このフィールドの使用は避けたほうがよいでしょう。

getpwuid関数とgetpwnam関数は、いずれも指定されたユーザーに対応するpasswd構造体へのポインタを返します。ユーザーは、getpwuidの場合にはUID、getpwnamの場合にはログイン名によって指定します。

例題 ユーザー情報

次のプログラムuser.cは、パスワードデータベースからユーザー情報を取り出します。

```
#include <sys/types.h>
#include <pwd.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
  uid_t uid;
  gid_t gid;
  struct passwd *pw;

  uid = getuid();
  gid = getgid();

  printf("User is %s\n", getlogin());

  printf("User IDs: uid=%d, gid=%d\n", uid, gid);

  pw = getpwuid(uid);
  printf("UID passwd entry:\n name=%s, uid=%d, gid=%d, home=%s, shell=%s\n",
       pw->pw_name, pw->pw_uid, pw->pw_gid, pw->pw_dir, pw->pw_shell);

  pw = getpwnam("root");
  printf("root passwd entry:\n");
  printf("name=%s, uid=%d, gid=%d, home=%s, shell=%s\n",
       pw->pw_name, pw->pw_uid, pw->pw_gid, pw->pw_dir, pw->pw_shell);
  exit(0);
}
```

プログラムを実行すると、次のように出力されます。出力される内容は、システムによって多少異なります。

```
$ user
User is neil
User IDs: uid=501, gid=100
UID passwd entry:
 name=neil, uid=501, gid=100, home=/home/neil, shell=/bin/bash
root passwd entry:
name=root, uid=0, gid=0, home=/root, shell=/bin/bash
```

解説

このプログラムでは、まずgetuidを呼び出して現在のユーザーのUIDを取得します。次に、取得したUIDを使って詳細なパスワードファイル情報を取得します。また、ユーザー名にroot

を指定して**getpwnam**関数を呼び出し、**root**のユーザー情報を表示します。

メモ

getuidの使い方については、idコマンドなどのソースコードが参考になります。

パスワードファイルのエントリを順次取り出す**getpwent**関数を使うと、パスワードファイル情報をすべて走査することができます。

```
#include <pwd.h>

void endpwent(void);
struct passwd *getpwent(void);
void setpwent(void);
```

getpwent関数は、各ユーザー情報を順に返します。エントリが残っていない場合には、**NULL**を返します。必要なエントリを走査したあとには、**endpwent**関数を使って処理を終了します。**setpwent**関数は、次の**getpwent**の呼び出し時に走査を開始するパスワードファイル内の位置を再設定します。これらの関数は、第3章で説明したディレクトリ走査関数**opendir**、**readdir**、**closedir**とほぼ同じように動作します。

4.5.1 その他のユーザー情報関数

ユーザーIDとグループID（実効IDと実ID）を取得するには、次の関数を使います。

```
#include <sys/types.h>
#include <unistd.h>

uid_t geteuid(void);
gid_t getgid(void);
gid_t getegid(void);
int setuid(uid_t uid);
int setgid(gid_t gid);
```

グループIDと実効ユーザーIDを操作する必要はあまりないでしょう。詳細については、マニュアルページを参照してください。

注意

setuidと**setgid**を呼び出すことができるのは、スーパーユーザーだけです。

4.6 ホスト情報

ユーザー情報同様、プログラムが実行されているコンピュータに関する情報もプログラムから取得することができます。**uname(1)**コマンドを使うとこのような情報が表示されますが、システムコールとして用意されている**uname(2)**を使うと、Cプログラムの中から同じ情報を取得することができます。

ホスト情報は、さまざまな場面で活用することができます。たとえば、学生用のマシンと管理者用のマシンを区別するなど、マシンのネットワーク上の名前によってプログラムの動作をカスタマイズすることもできます。また、ライセンスを管理する場合にも、1つのマシンだけで実行できるようにプログラムを制限することが可能です。

UNIXシステムにネットワークコンポーネントがインストールされている場合は、**gethostname**関数を呼び出すことでマシンのネットワーク名を簡単に取得できます。

```
#include <unistd.h>

int gethostname(char *name, size_t namelen);
```

gethostname関数は、マシンのネットワーク名を文字列**name**に書き込みます。文字列**name**は、少なくとも**namelen**文字の長さを持つものとして扱われます。**gethostname**は成功すると0を返し、それ以外の場合には-1を返します。

ホストコンピュータに関してさらに詳しい情報を得るには、**uname**システムコールを使います。

```
#include <sys/utsname.h>

int uname(struct utsname *name);
```

uname関数は、パラメータ**name**で示される構造体にホスト情報を書き込みます。**utsname**構造体は**sys/utsname.h**で定義されており、少なくとも次のメンバを持っています。

表4.4 utsname構造体のメンバ

メンバ	説明
char sysname[]	オペレーティングシステム名
char nodename[]	ホスト名
char release[]	システムのリリースレベル
char version[]	システムのバージョン番号
char machine[]	ハードウェアの種類

unameは、成功すると0を返します。それ以外の場合には-1を返し、**errno**にエラーの内容を示す値を設定します。

例題 ホスト情報

次のプログラムhostget.cは、ホストコンピュータについての情報を取得します。

```
#include <sys/utsname.h>
#include <unistd.h>
#include <stdio.h>

int main()
{
  char computer[256];
  struct utsname uts;

  if(gethostname(computer, 255) != 0 || uname(&uts) < 0) {
    fprintf(stderr, "Could not get host information\n");
    exit(1);
  }

  printf("Computer host name is %s\n", computer);
  printf("System is %s on %s hardware\n", uts.sysname, uts.machine);
  printf("Nodename is %s\n", uts.nodename);
  printf("Version is %s, %s\n", uts.release, uts.version);
  exit(0);
}
```

Linuxマシンでプログラムを実行すると、次のように出力されます。

```
$ hostget
Computer host name is tilde
System is Linux on i586 hardware
Nodename is tilde
Version is 2.0.35, #1 Tue Oct 13 04:01:38 JST 1998
```

解説

このプログラムでは、まずgethostnameを呼び出してホストコンピュータのネットワーク名を取得します。実行例では、tildeという名前が返されています。次にunameを呼び出し、このマシンに関する詳しい情報を取得します。unameが返す文字列の形式は、システムに依存します。実行例では、バージョン文字列にはカーネルがコンパイルされた日付が含まれています。

メモ uname(2)の使い方については、uname(1)コマンドのソースコードが参考になります。

4.6.1 ライセンス

gethostid関数を使うと、各ホストコンピュータが持つ一意の識別子を取得することができます。

```
#include <unistd.h>

long gethostid(void);
```

gethostidは、ホストコンピュータが持つ一意の値を返す関数です。ライセンスマネージャなどでは、この情報を使うことで、プログラムの実行可能なマシンを、有効なライセンスを持つマシンだけに制限することができます。Sunワークステーションでは、コンピュータの製造時に不揮発メモリに設定された値を返すので、この値はシステムハードウェア固有の値になります。

Linuxなど、その他のシステムでは、マシンのインターネットアドレスに基づいて値を返すので、ライセンス管理に使用できるだけの十分な一意性は保証されません。

4.7 ログ

アプリケーションでは、しばしば動作状況を記録する必要があります。また、システムプログラムもコンソールにメッセージを書き込んだり、ログファイルにメッセージを記録したりします。これらのメッセージは、エラーや警告、あるいはシステムの状態に関する一般的な情報を伝えるために使われます。たとえば、**su**プログラムでは、ユーザーがスーパーユーザーの権限を得ようとして失敗した場合に、そのことを記録するようになっています。

ほとんどの場合、これらのログメッセージは、特別なディレクトリに置かれたシステムファイルに記録されます。ディレクトリのある場所は、**/usr/adm**または**/var/log**が一般的です。典型的なLinuxシステムの場合、**/var/log/messages**にはすべてのシステムメッセージが、**/var/log/syslog**にはその他のログメッセージが、**/var/log/debug**にはデバッグメッセージが記録されます。システムの設定がどうなっているかは、ファイル**/etc/syslog.conf**で確かめることができます。次に示すのはメッセージの例です。

```
Nov  4 09:41:21 tilde kernel: IP Protocols: ICMP, UDP, TCP
Nov  4 09:41:21 tilde kernel: ide0 at 0x1f0-0x1f7,0x3f6 on irq 14
Nov  4 09:41:21 tilde kernel: Floppy drive(s): fd0 is 1.44M
Nov  4 09:41:21 tilde kernel: FDC 0 is a post-1991 82077
Nov  4 09:41:23 tilde sendmail[59]: starting daemon (8.7.5+2.6Wbeta6):
SMTP+queueing@1+00:00:00
Nov  4 09:41:33 tilde login[60]: ROOT LOGIN on `tty1'
```

最初の数行は、起動後にハードウェアの検出などを行っているLinuxカーネルから出力されたメッセージです。次に**sendmail**の起動メッセージが記録されています。最後は**login**プログラムからのメッセージで、スーパーユーザーがログインしたことを示しています。

一部のUNIXシステムでは、この例のような人間が読める形でのメッセージファイルを提供していないことがあります。こうしたシステムでは、システムイベントが収められたデータベースを読むためのツールが用意されています。詳細については、システムのマニュアルを参照してください。

システムメッセージの形式と格納方法はシステムによって異なりますが、メッセージを生成する方法には標準があります。UNIXの仕様では、すべてのプログラムから利用可能なログメッセージ生成インタフェースとして、**syslog**関数を使用することができます。

```
#include <syslog.h>

void syslog(int priority, const char *message, ...);
```

syslog関数は、ログメッセージをログファシリティに送ります。引数**priority**には、各メッセージのプライオリティレベルとファシリティの値とのビット単位の論理和を指定します。プライオリティレベルは、ログメッセージをどのように扱うかを示すものです。ファシリティの値は、メッセージの送信元を記録するためのものです。

ファシリティの値は**syslog.h**で定義されており、ユーザーアプリケーションから送られたメッセージであることを示す**LOG_USER**（デフォルト）、ローカル管理者がしかるべき意味を与えて使用できる**LOG_LOCAL0**から**LOG_LOCAL7**までの値などが用意されています。

プライオリティレベルを優先順位の高いものから示すと、次の表のようになります。

表4.5 syslog関数のプライオリティレベル

プライオリティレベル	説明
LOG_EMERG	緊急事態
LOG_ALERT	データベースの破損など、プライオリティの高い問題の発生
LOG_CRIT	ハードウェアエラーなどのクリティカルなエラー
LOG_ERR	エラー
LOG_WARNING	警告
LOG_NOTICE	注意の必要な状態
LOG_INFO	情報メッセージ
LOG_DEBUG	デバッグメッセージ

システムの設定によっては、**LOG_EMERG**メッセージをすべてのユーザーにブロードキャストし、**LOG_ALERT**メッセージについては管理者にメールを送信、**LOG_DEBUG**メッセージは無視、それ以外のメッセージはメッセージファイルに記録するようになっていることもあります。ログ機能を利用するプログラムは簡単に作成できます。必要な操作は、ログメッセージを作成したいときに**syslog**関数を呼び出すだけです。

syslogによって作成されるログメッセージは、メッセージヘッダーとメッセージ本体から構成されます。メッセージヘッダーは、ファシリティおよび日付と時刻から作成されます。メッセージ本体は、**syslog**関数のパラメータ**message**から作成されます。**message**は、**printf**の書式指定文字列と同じような働きをします。**syslog**関数に指定する追加の引数は、文字列**messege**の中に埋め込まれた**printf**に似た変換指定子によって変換されます。このとき、**%m**という変換指定子を使うと、エラー変数**errno**の現在の値に関連付けられたエラーメッセージの文字列がログメッセージに挿入されます。**%m**は、エラーメッセージを記録する場合に使うと便利です。

例題 syslog

次のプログラム**syslog.c**は、存在しないファイルを開き、意図的にエラーを発生させます。

```
#include <syslog.h>
#include <stdio.h>

int main()
{
  /* Try opening a non-existent file */
  FILE *f;

  f = fopen("not_here","r");
  if(!f)
    syslog(LOG_ERR|LOG_USER,"oops - %m\n");
  exit(0);
}
```

このプログラムをコンパイルして実行すると、画面にはなにも出力されませんが、ファイル**/var/log/syslog**の最後に次のようなメッセージが記録されます。

```
Nov  6 18:13:06 tilde syslog: oops - No such file or directory
```

解説

このプログラムでは、存在しないファイルをオープンしようとしています。オープンに失敗した場合には、**syslog**を呼び出してシステムのログにメッセージを出力します。

ログメッセージには、どのプログラムがログファシリティを呼び出したかは記録されておらず、**syslog**がメッセージの指定とともに呼び出されたことだけが記録されています。また、変換指定子**%m**がエラーを説明する文字列に置き換えられており、エラー番号だけを出力する場合と比べていくらか親切なメッセージになっています。

4.7.1 ログの設定

syslog.hでは、ログファシリティの動作を変更するときに使うその他の関数も定義されています。これらの関数の構文は次のとおりです。

```
#include <syslog.h>

void closelog(void);
void openlog(const char *ident, int logopt, int facility);
int setlogmask(int maskpri);
```


openlog関数を呼び出すと、ログメッセージの形式を変更することができます。パラメータ**ident**には、ログメッセージの前に付加する文字列を指定します。**ident**を使えば、どのプログラムがメッセージを作成したのかを記録することができます。パラメータ**facility**には、以後の**syslog**の呼び出しで使うデフォルトのファシリティの値を指定します。この値のデフォルトは、**LOG_USER**です。**logopt**は、以後の**syslog**の呼び出し時の動作を設定するパラメータで、次の表に示したものから0個以上のパラメータをビット単位の論理和で指定します。

表4.6 openlog関数のlogoptパラメータ

パラメータ	説明
LOG_PID	プロセスID（システムが各プロセスに割り当てる一意の番号）をメッセージに含める。
LOG_CONS	メッセージを記録できない場合、コンソールにメッセージを送る。
LOG_ODELAY	syslogが初めて呼び出された時点でログファシリティをオープンする。
LOG_NDELAY	実際にログが行われる時点まで待たず、ただちにログファシリティをオープンする。

openlog関数は、ファイルデスクリプタを割り当ててオープンし、このファイルデスクリプタを使ってログファシリティへの書き込みを行います。**closelog**関数を呼び出すと、ファイルデスクリプタをクローズすることができます。**syslog**関数は必要に応じて自分でログファシリティをオープンするので、**syslog**を呼び出す前に**openlog**関数を呼び出す必要はありません。

setlogmaskを使ってログマスクを設定すると、ログメッセージのプライオリティレベルを制御することができます。ログマスクの設定を行うと、ログマスクで設定されていないプライオリティレベルを持つ、以後のすべての**syslog**の呼び出しは無視されます。このことを利用すれば、プログラムの本体を変更せずに**LOG_DEBUG**メッセージを無効にしたりすることが可能です。

ログメッセージのマスクを作成するには、**LOG_MASK(priority)**または**LOG_UPTO(priority)**を使います。**LOG_MASK(priority)**を使うと、1つのプライオリティレベルだけからなるマスクが作成されます。**LOG_UPTO(priority)**を使うと、指定されたプライオリティまでの（指定されたプライオリティを含む）すべてのプライオリティからなるマスクが作成されます。

例題 logmask

次のプログラムlogmask.cは、logmaskの使い方の一例です。

```
#include <syslog.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
  int logmask;

  openlog("logmask", LOG_PID|LOG_CONS, LOG_USER);
  syslog(LOG_INFO,"informative message, pid = %d", getpid());
  syslog(LOG_DEBUG,"debug message, should appear");
  logmask = setlogmask(LOG_MASK(LOG_NOTICE));
  syslog(LOG_DEBUG,"debug message, should not appear");
  exit(0);
}
```

このプログラムを実行しても画面にはなにも表示されませんが、Linuxシステムでは、/var/log/messageの最後に次のようなメッセージが記録されます。

```
Nov  6 20:58:59 tilde logmask[26221]: informative message, pid = 26221
```

さらに、/var/log/debugには次のようなメッセージが記録されます。

```
Nov  6 20:58:59 tilde logmask[26221]: debug message, should appear
```

解説

このプログラムでは、プログラム名のlogmaskを指定してログファシリティを初期化し、ログメッセージにプロセスIDを含めるようにしています。次に、syslogを呼び出しています。情報メッセージは/var/log/messagesに記録され、デバッグメッセージは/var/log/debugに記録されます。しかし、2つ目のデバッグメッセージは記録されません。これは、その直前にsetlogmaskを呼び出し、LOG_NOTICEよりプライオリティの低いすべてのメッセージを無視するようにしているからです（古いバージョンのLinuxカーネルでは、説明どおりに動作しないかもしれません）。

logmask.cではgetpid関数を使っています。getpid関数、およびこの関数と関連の深いgetppid関数の構文は次のとおりです。

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void);
pid_t getppid(void);
```

これらの関数は、それぞれ関数を呼び出したプロセスのプロセスIDと親プロセスIDを返します。プロセスIDについては、第10章で詳しく取り上げます。

4.8 リソースと制限

UNIXシステムで実行されるプログラムには、リソースに関する種々の制限が課せられています。このような制限としては、ハードウェアによる物理的制限（メモリなど）、システムポリシーによる制限（CPU時間の上限など）、あるいは実装上の制限（`int`のサイズ、ファイル名に使用できる文字数など）があります。このうちいくつかの制限は、UNIXの仕様で定義されており、その内容をアプリケーションから取得することができます。第7章では、こうした制限や、制限を破った場合にどのような結果を生じるかについて詳しく取り上げます。

ヘッダーファイル`limits.h`では、オペレーティングシステムによって課せられた制約を表すための定数が定義されています。次の表は、そのごく一部を示したものです。

表4.7　オペレーティングシステムによる制約

定数	説明
`NAME_MAX`	ファイル名に使用できる最大文字数
`CHAR_BIT`	`char`のビット数
`CHAR_MAX`	`char`の最大値
`INT_MAX`	`int`の最大値

アプリケーションで使う可能性のある定数は、これ以外にも数多くあります。システムにインストールされているヘッダーファイルを一度覗いておくとよいでしょう。`NAME_MAX`はファイルシステムに固有の値なので、移植性を考慮してコードを記述するときには`pathconf`関数を使います。詳細については、`pathconf`関数のマニュアルページを参照してください。

ヘッダーファイル`sys/resource.h`には、リソースの操作に関連するさまざまな定義が記述されています。また、プログラムで許容されているサイズ、実行優先度、ファイルリソースなどに関する制限を取得または設定するための関数も定義されています。

```
#include <sys/resource.h>

int getpriority(int which, id_t who);
int setpriority(int which, id_t who, int priority);
int getrlimit(int resource, struct rlimit *r_limit);
int setrlimit(int resource, const strcut rlimit *r_limit);
int getrusage(int who, struct rusage *r_usage);
```

id_tは、ユーザーIDとグループID用に使われる整数型です。構造体**rusage**は**sys/resource.h**で定義されており、現在のプログラムがどれだけのCPU時間を使用しているかを取得するために使われます。**rusage**構造体は、少なくとも次のメンバを含んでいなければなりません。

表4.8 rusage構造体のメンバ

メンバ	説明
struct timeval ru_utime	使用したユーザー時間
struct timeval ru_stime	使用したシステム時間

timeval構造体は**sys/time.h**で定義されており、**tv_sec**フィールドと**tv_usec**フィールドを含んでいます。これらのフィールドは、それぞれ秒とマイクロ秒を表します。

プログラムが消費するCPU時間は、**ユーザー時間**と**システム時間**に分かれます。ユーザー時間は、プログラムが自身の命令の実行に費やした時間です。システム時間は、プログラムに代わってオペレーティングシステムが消費した時間、つまり入出力やその他の機能を実行するシステムコールに費やされた時間を表します。

getrusage関数は、CPU時間に関する情報をパラメータ**r_usage**で示される**rusage**構造体に書き込みます。パラメータ**who**には、次のいずれかの定数を指定します。

表4.9 getrusage関数のwhoパラメータ

定数	説明
RUSAGE_SELF	現在のプログラムに関する使用情報のみを返す。
RUSAGE_CHILDREN	子プロセスに関する使用情報も含める。

子プロセスとタスクの優先度については第10章で詳しく取り上げます。ここではとりあえず、各プログラムにはそれぞれある優先度が関連付けられており、優先度が高いプログラムほど、利用可能なCPU時間を多く割り当てられるということを覚えておいてください。一般ユーザーはプログラムの優先度を引き下げることはできますが、引き上げることはできません。

アプリケーションでは、**getpriority**関数と**setpriority**関数を使って自身（およびほかのプログラム）の優先度を調べたり、設定したりすることができます。これらの関数を使って優先度を取得または設定するプロセスは、プロセスID、グループID、ユーザーのいずれかで指定します。このパラメータ**who**をどのように扱うかを指定するのがパラメータ**which**で、次のいずれかを指定します。

表4.10 getpriority／setpriority関数のwhichパラメータ

パラメータ	説明
PRIO_PROCESS	whoはプロセスID
PRIO_PGRP	whoはプロセスグループ
PRIO_USER	whoはユーザー

たとえば、現在のプロセスの優先度を取得するには、次のように記述します。

```
priority = getpriority(PRIO_PROCESS, getpid());
```


setpriority関数は、要求された操作が可能であれば新しい優先度を設定します。

デフォルトの優先度は0です。正の優先度は、より高い優先度を持つほかの実行可能なタスクが存在しないときに実行されるバックグラウンドタスクに対して使います。負の優先度を持つプログラムは、より頻繁に実行され、利用可能なCPU時間もそれだけ多く使用します。有効な優先度の範囲は-20～+20です。数値が大きいほど優先度が低くなる点に注意してください。

getpriority関数は、成功すると有効な優先度を返します。エラーが発生した場合には-1を返し、errnoにエラーの内容を示す値を設定します。エラーの際に返される-1自体も有効な優先度なので、getpriorityを呼び出す場合には、あらかじめerrnoに0をセットし、関数から戻った時点でerrnoが0のままかどうかを調べる必要があります。setpriority関数は成功すると0を返し、失敗すると-1を返します。

システムリソースに関する制限を取得または設定するには、getrlimit関数とsetrlimit関数を使います。どちらの関数も、リソースの制限を表すための汎用的な構造体rlimitを使います。rlimit構造体はsys/resource.hで定義されており、次のメンバを持っています。

表4.11 rlimit構造体のメンバ

メンバ	説明
rlim_t rlim_cur	現在のソフト制限
rlim_t rlim_max	ハード制限

定義済みの型rlim_tは、リソースのレベルを記述するのに使う整数型です。通常、**ソフト制限**は超えるべきではない制限を意味し、この制限を超えると、ライブラリ関数がエラーを返すことがあります。**ハード制限**を超えた場合には、システムがプログラムにシグナルを送り、プログラムを終了させようとします。シグナルの例としては、CPU時間の制限を超えた場合に送られるSIGXCPU、データサイズの制限を超えた場合に送られるSIGSEGVなどがあります。プログラムでは、自身のソフト制限をハード制限より少ない任意の値に設定することができます。また、ハード制限を引き下げることも可能です。ただし、ハード制限を引き上げることができるのは、スーパーユーザー権限で実行されるプログラムだけです。

制限の可能なシステムリソースはいくつか存在します。これらはsys/resource.hで次のよう

に定義されており、**rlimit**関数群のパラメータ**resource**で指定します。

表4.12 rlimit関数群のresourceパラメータ

パラメータ	説明
RLIMIT_CORE	コアダンプファイルサイズの制限（バイト単位）
RLIMIT_CPU	CPU時間の制限（秒単位）
RLIMIT_DATA	データ（malloc／sbrk）セグメントの制限（バイト単位）
RLIMIT_FSIZE	ファイルサイズの制限（バイト単位）
RLIMIT_NOFILE	オープンするファイル数の制限
RLIMIT_STACK	スタックサイズの制限（バイト単位）
RLIMIT_AS	アドレス空間（スタックおよびデータ）の制限（バイト単位）

次に示す**limits.c**は、典型的なアプリケーションの動作をごく大まかにシミュレートしたプログラムです。プログラムの中でリソースの制限を設定し、制限を破っています。

例題 リソースの制限

1 まず、プログラムで使用する関数のインクルードファイルを記述します。

```
#include <sys/types.h>
#include <sys/resource.h>
#include <sys/time.h>
#include <unistd.h>
#include <stdio.h>
#include <math.h>
```

2 **work**関数は、文字列を作業ファイルに10000回書き込んだあと、さらにCPUに負荷をかけるために算術演算を実行します。

```
void work()
{
    FILE *f;
    int i;
    double x = 4.5;

    f = tmpfile();
    for(i = 0; i < 10000; i++) {
        fprintf(f,"Do some output\n");
        if(ferror(f)) {
            fprintf(stderr,"Error writing to temporary file\n");
            exit(1);
        }
    }
    for(i = 0; i < 100000; i++)
        x = log(x*x + 3.21);
}
```

3 mainで関数では、work関数を呼び出したあと、getrusage関数を使って消費したCPU時間を調べます。また、得られた結果を画面に表示します。

```
int main()
{
    struct rusage r_usage;
    struct rlimit r_limit;
    int priority;

    work();
    getrusage(RUSAGE_SELF, &r_usage);

    printf("CPU usage: User = %ld.%06ld, System = %ld.%06ld\n",
        r_usage.ru_utime.tv_sec, r_usage.ru_utime.tv_usec,
        r_usage.ru_stime.tv_sec, r_usage.ru_stime.tv_usec);
```

4 次に、getpriorityとgetrlimitを呼び出して現在の優先度とファイルサイズの制限を調べます。

```
    priority = getpriority(PRIO_PROCESS, getpid());
    printf("Current priority = %d\n", priority);

    getrlimit(RLIMIT_FSIZE, &r_limit);
    printf("Current FSIZE limit: soft = %ld, hard = %ld\n",
        r_limit.rlim_cur, r_limit.rlim_max);
```

5 最後に、setrlimitを使ってファイルサイズの制限を設定し、もう一度work関数を呼び出します。しかし、制限を超えるサイズのファイルを作成しようとするためにwork関数は失敗します。

```
    r_limit.rlim_cur = 2048;
    r_limit.rlim_max = 4096;
    printf("Setting a 2K file size limit\n");
    setrlimit(RLIMIT_FSIZE, &r_limit);

    work();
    exit(0);
}
```

このプログラムを実行すると、使用されたCPU資源とプログラムの実行中のデフォルトの優先度が表示されます。ファイルサイズの制限を設定したあとは、一時ファイルに2048バイトを超える書き込みを行うことができなくなります。

```
$ gcc -o limits limits.c -lm
$ limits
CPU usage: User = 0.130000, System = 0.000000
Current priority = 0
```

```
Current FSIZE limit: soft = 2147483647, hard = 2147483647
Setting a 2K file size limit
Error writing to temporary file
```

niceコマンドを使ってプログラムを起動すれば、プログラムの優先度を変更することができます。次の例では優先度を+10に変更しています。その結果、プログラムの実行にはより多くの時間がかかるようになります。

```
$ nice limits
CPU usage: User = 0.110000, System = 0.020000
Current priority = 10
Current FSIZE limit: soft = 2147483647, hard = 2147483647
Setting a 2K file size limit
Error writing to temporary file
```

解説

limitsプログラムは、典型的なプログラムの動作をシミュレートする**work**関数を呼び出します。このプログラムでは150Kほどのデータを一時ファイルに書き込み、計算を行います。その後、適切な関数を呼び出してプログラムの優先度とファイルサイズの制限を調べます。出力結果からわかるとおり、ファイルサイズの制限は最大値に設定されており、ディスクに空きがあれば2GBのファイルを作成することができます。次に、このファイルサイズの制限を2Kに設定し、もう一度**work**関数を呼び出します。今度は、制限を超える大きなサイズの一時ファイルを作成できず、処理は失敗します。

メモ

bashの**ulimit**コマンドを使うと、個々のシェルから実行するプログラムに制限を設定することができます。

4.9 この章のまとめ

この章では、UNIXの環境について取り上げ、プログラムの実行時のさまざまな条件について説明しました。コマンドライン引数や環境変数を使えば、プログラムのデフォルトの動作を変更したり、プログラムに有益なオプションを用意したりすることができます。

また、ライブラリ関数を使って日付と時刻の値を操作したり、プログラムやユーザー、さらにプログラムが実行されているコンピュータに関する情報を取得する方法を示しました。

UNIXプログラムは貴重なリソースを共有しなければなりません。この章では、使用できるリソースを取得したり管理したりする方法も示しました。

第5章

端末

第2章の最後では、CDデータベースアプリケーションを作成しました。このアプリケーションは正しく動作するものの、改善すべき点も多くあります。たとえば、ユーザーインタフェースをもっと洗練されたものにすれば、操作感は大幅に向上します。

この章では、ユーザーの端末、つまりキーボード入力と画面出力を細かく制御する方法について説明します。具体的には、端末から受け取ったさまざまなユーザー入力をプログラムに渡す方法、プログラムから画面の適切な位置に出力を送る方法を示します。

実際に改良を加えたCDデータベースアプリケーションは、次の章を終えるまで完成しませんが、この章ではあとで必要になる基礎知識とテクニックを中心に、端末入出力の考え方をUNIXの哲学とからませながら説明します。次の第6章では、端末の画面表示を制御するための関数ライブラリ**curses**を取り上げます。この章のテーマは端末に対する低水準アクセスですが、ここで得た知識は今後もいろいろな場面で役立つはずです。

取り上げる項目は次のとおりです。

- 端末に対する読み取りと書き込み
- 端末ドライバと汎用端末インタフェース
- **termios**
- 端末出力と**terminfo**
- キーストロークの検出

5.1 端末に対する読み取りと書き込み

第3章で説明したように、コマンドプロンプトからプログラムを起動すると、シェルが入力ストリームと出力ストリームをプログラムに結び付けます。ということは、**getchar**と**printf**を使ってこれらのデフォルトストリームに対する読み取りと書き込みを行えば、ユーザーとの対話も可能になるはずです。

さっそく、**getchar**と**printf**を使ってCDデータベースアプリケーションのメニュールーチンをCで書き直してみましょう。ここでは、プログラムの名前を**menu1.c**としておきます。

例題 Cによるメニュールーチン

1 まず、メニューを配列として定義し、**getchoice**関数のプロトタイプを宣言します。

```
#include <stdio.h>

char *menu[] = {
    "a - add new record",
    "d - delete record",
```

```
    "q - quit",
    NULL,
};

int getchoice(char *greet, char *choices[]);
```

2 mainの関数では、定義した変数menuを指定してgetchoiceを呼び出します。

```
int main()
{
    int choice = 0;

    do
    {
        choice = getchoice("Please select an action", menu);
        printf("You have chosen: %c\n", choice);
    } while (choice != 'q');
    exit(0);
}
```

3 次は、このプログラムの中心となるgetchoice関数です。この関数はメニューを表示し、ユーザーからの入力を読み取ります。

```
int getchoice(char *greet, char *choices[])
{
    int chosen = 0;
    int selected;
    char **option;

    do {
        printf("Choice: %s\n",greet);
        option = choices;
        while(*option) {
            printf("%s\n",*option);
            option++;
        }
        selected = getchar();
        option = choices;
        while(*option) {
            if(selected == *option[0]) {
                chosen = 1;
                break;
            }
            option++;
        }
        if(!chosen) {
            printf("Incorrect choice, select again\n");
        }
    } while(!chosen);
    return selected;
}
```

解説

getchoiceは、**greet**で指定されたメッセージと**choices**で指定されたメニューを表示したあと、ユーザーにメニューの先頭の文字を入力するように求めます。プログラムは、**getchar**が配列**option**のエントリの先頭の1文字に一致する文字を返すまでループします。

プログラムをコンパイルして実行すると、どうも期待したとおりには動作しないようです。実行例を次に示します。

```
$ menu1
Choice: Please select an action
a - add new record
d - delete record
q - quit
a
You have chosen: a
Choice: Please select an action
a - add new record
d - delete record
q - quit
Incorrect choice, select again
Choice: Please select an action
a - add new record
d - delete record
q - quit
q
You have chosen: q
$
```

この実行例では、メニューを選択するのに、**a**、**Return**、**q**、**Return**と入力する必要がありました。問題は少なくとも2つあります。ひとつは、正しい選択をした場合でも必ず**Incorrect choice**というメッセージが表示されることです。もうひとつは、**Return**キーを入力しないとプログラムが入力を読み取らない点です。

問題点

上に指摘した2つの問題は、実は互いに関連し合っています。まず、端末入力はユーザーが**Return**キーを押すまでプログラムからは利用できません。ほとんどの場合、これはむしろ便利な機能といえます。ユーザーは**BackSpace**キーや**Delete**キーを使って入力内容を修正し、画面に表示されている内容で間違いないことを確認した時点で**Return**キーを押せばよいからです。

このような動作は、**カノニカルモード**（canonical mode）または**標準モード**と呼ばれます。このモードでは、入力はすべて行単位で処理されます。1つの入力行が完結するまでは（通常はユーザーが**Return**キーを押すまで）、端末インタフェースが**BackSpace**なども含めてすべてのキー入力を管理し、アプリケーションから文字を読み取ることはできません。

これに対して、**非カノニカルモード**（non-canonical mode）では、入力された文字の処理をア

プリケーションからいろいろと制御することができます。これら2つのモードについては、あとでもう一度取り上げます。

UNIXの端末ハンドラで特筆すべきことは、割り込み文字がシグナルに変換され、ユーザーに代わって**BackSpace**キーや**Delete**キーが自動的に処理される点です。つまり、各プログラムでこれらの処理を実装する必要はありません。シグナルについては、第10章で詳しく解説します。

さて、上で作成したプログラムに戻ってどこが問題だったのか、もう少し具体的に考えてみましょう。入力した文字は、ユーザーが**Return**キーを押すまでプログラムに渡されません。しかし、ユーザーが**Return**キーを押すと、入力した文字と**Return**キーの両方がプログラムに渡されます。したがって、ユーザーがメニューを選択すると、プログラムは`getchar`を呼び出し、入力した文字を処理したあと、再び`getchar`を呼び出して**Return**キーを処理してしまいます。

このときプログラムが受け取る文字は、ASCIIのキャリッジリターン（`CR`、16進コードの`0x0d`）ではなく、ラインフィード（`LF`、16進コードの`0x0a`）です。これは、UNIXでは内部的に行の終わりを示すのに常にラインフィードが使われ、ラインフィード単独で改行を意味するからです。この点は、キャリッジリターンとラインフィードを組み合わせて使うDOSとは異なります。入力デバイスや出力デバイスでキャリッジリターンが使われている場合には、UNIXの端末処理がその面倒を見ます。DOSやその他の環境に慣れている場合には、こうした動作に少し違和感を覚えるかもしれませんが、これは実はUNIXの非常に大きな利点です。というのも、テキストファイルとバイナリファイルを実質的に区別する必要がなくなるからです。キャリッジリターンは、端末や一部のプリンタ、プロッタなどとの間で入出力を行う場合にのみ処理されます。

メニューを選択する文字に続けて送られるラインフィードの問題を回避する方法としては、次のコードのようにラインフィードを無視するやり方があります。

```
do {
        selected = getchar();
} while(selected == '\n');
```

これで、最初の問題はとりあえず回避できます。もうひとつの問題、つまりメニューを選択するたびに**Return**キーを押す必要がある点については、ラインフィードの扱い方も含めて、少しあとでエレガントな解決方法を示します。

5.1.1 リダイレクトされた出力の扱い

UNIXプログラムでは、対話的なプログラムの場合でも、頻繁に入力や出力をファイルやほかのプログラムにリダイレクトします。さっそく、上で作成したプログラムの出力もファイルにリダイレクトしてみましょう。

```
$ menu1 > file
a
q
$
```

たしかに出力は端末ではなくファイルにリダイレクトされています。しかし、これではプログラムからのメッセージも表示されません。リダイレクトしても問題のない出力だけをリダイレクトし、ユーザーに表示したいメッセージは端末に送ることができれば、ユーザーインタフェースとしてもより望ましいはずです。

標準出力がリダイレクトされているかどうかは、低水準ファイルデスクリプタが端末に関連付けられているかどうかを調べることで判断できます。それには、**isatty**システムコールを使います。**isatty**は、指定された有効なファイルデスクリプタが端末に接続されているかどうかを調べ、その結果を返します。

```
#include <unistd.h>

int isatty(int fildes);
```

isattyは、**fildes**で指定されたオープンされているファイルデスクリプタが端末に接続されている場合は1を返し、それ以外の場合には0を返します。

メニューを表示するプログラムではファイルストリームを使っていますが、**isatty**の引数に指定できるのはファイルデスクリプタです。このため、**isatty**を呼び出すときは、第3章で説明した**fileno**ルーチンを組み合わせて使う必要があります。

stdoutがリダイレクトされていた場合には、どのように対処すればよいでしょうか。単にプログラムを終了するのはあまりよい方法とはいえません。なぜプログラムが動作しないのか、ユーザーにはわからないからです。**stdout**にメッセージを表示する方法も、**stdout**自体がリダイレクトされているので使用できません。ひとつの方法は、**stderr**にメッセージを書き込むことです。この方法なら、シェルで**> file**と指定してもメッセージはリダイレクトされません。

例題 出力がリダイレクトされているかどうかのチェック

1 最初に作成したプログラム**menu1.c**の**main**関数を変更し、**menu2.c**を作成します。インクルードするファイルが1つ増えている点に注意してください。

```
#include <unistd.h>
...
int main()
{
    int choice = 0;

    if(!isatty(fileno(stdout))) {
        fprintf(stderr,"You are not a terminal!\n");
```

```
            exit(1);
        }

        do {
            choice = getchoice("Please select an action", menu);
            printf("You have chosen: %c\n", choice);
        } while (choice != 'q');
        exit(0);
    }
```

2 プログラムを実行します。

```
$ menu2
Choice: Please select an action
a - add new record
d - delete record
q - quit
q
You have chosen: q
$ menu2 > file
You are not a terminal!
$
```


解説

新しい**main**関数では、**isatty**関数を使って標準出力が端末に接続されているかどうかを調べ、リダイレクトされている場合にはプログラムの実行を中止します。シェルも同じようなしくみを使ってプロンプトを表示するかどうかを決定しています。**stdout**と**stderr**の両方をリダイレクトすることも可能で、この方法もよく使われます。また、次のように指定して実行すれば、エラーストリームだけを別のファイルにリダイレクトすることができます。

```
$ menu2 >file 2>file.error
$
```

2つの出力ストリームを両方とも1つのファイルにリダイレクトするには、次のようにします。

```
$ menu2 >file 2>&1
$
```

このように出力が両方ともリダイレクトされている場合には、メッセージをコンソールに送る必要があります。（出力のリダイレクトについては第2章で取り上げました。構文についても詳しく説明していますから、上のようなリダイレクトのしくみがよくわからない場合には、第2章を読み直してください。）

5.1.2 端末とのやりとり

プログラムの中でユーザーと対話する部分をリダイレクトされないようにして、その他の入出力をリダイレクトできるようにするには、ユーザーと対話する部分をstdoutやstderrから切り離す必要があります。具体的には、端末に対して直接読み取りや書き込みを行うと、目的を実現できます。ただし、UNIXはマルチユーザーシステムなので、通常は複数の端末が直接またはネットワークを介して接続されています。では、どの端末を使用すべきかをどうやって判断したらよいのでしょうか。

さいわい、UNIXでは、常に現在の端末を意味する特殊デバイス/dev/ttyが用意されています。UNIXではすべてのものがファイルとして扱われるので、/dev/ttyに対しても通常のファイル操作を使って読み取りや書き込みを行うことができます。

メニュー選択プログラムを修正し、getchoiceルーチンにパラメータを渡して出力を細かく制御できるようにします。プログラムの名前はmenu3.cとします。

例題 /dev/ttyの使用

1 menu2.cを次のように修正し、入力と出力を/dev/ttyとの間でやりとりできるようにします。

```
#include <stdio.h>
#include <unistd.h>

char *menu[] = {
    "a - add new record",
    "d - delete record",
    "q - quit",
    NULL,
};
int getchoice(char *greet, char *choices[], FILE *in, FILE *out);
int main()
{
    int choice = 0;
    FILE *input;
    FILE *output;
    if (!isatty(fileno(stdout))) {
        fprintf(stderr,"You are not a terminal, OK.\n");
    }
    input = fopen("/dev/tty", "r");
    output = fopen("/dev/tty", "w");
    if(!input || !output) {
        fprintf(stderr,"Unable to open /dev/tty\n");
        exit(1);
    }
    do {
        choice = getchoice("Please select an action", menu, input, output);
```

```
        printf("You have chosen: %c\n", choice);
    } while (choice != 'q');
    exit(0);
}
int getchoice(char *greet, char *choices[], FILE *in, FILE *out)
{
    int chosen = 0;
    int selected;
    char **option;

    do {
        fprintf(out,"Choice: %s\n",greet);
        option = choices;
        while(*option) {
            fprintf(out,"%s\n",*option);
            option++;
        }
        do {
            selected = fgetc(in);
        } while (selected == '\n');
        option = choices;
        while(*option) {
            if(selected == *option[0]) {
                chosen = 1;
                break;
            }
            option++;
        }
        if(!chosen) {
            fprintf(out,"Incorrect choice, select again\n");
        }
    } while(!chosen);
    return selected;
}
```


出力をリダイレクトしてこのプログラムを実行すると、プログラムが表示するプロンプトとプログラムからの通常の出力が区別されて処理されることがわかります。

```
$ menu3 > file
You are not a terminal, OK.
Choice: Please select an action
a - add new record
d - delete record
q - quit
d
Choice: Please select an action
a - add new record
d - delete record
q - quit
q
$ cat file
You have chosen: d
You have chosen: q
```

5.2 端末ドライバと汎用端末インタフェース

プログラムで端末を使う場合、通常のファイル操作だけでは実現できない細かい制御が必要になることがあります。UNIXでは、端末ドライバの動作を制御するための一連のインタフェースが用意されており、これらのインタフェースを使うことで端末の入出力処理を細かく制御できるようになっています。

端末を制御するには、**汎用端末インタフェース**（GTI）と呼ばれる一連の関数呼び出しを使います。次の図に示すとおり、汎用端末インタフェースは、端末に対する読み書きのインタフェースとは別になっており、データを読み書きするためのインタフェースはすっきりとさせたまま、端末の動作を細かく制御できるようになっています。ただし、端末入出力インタフェースはさまざまなハードウェアを扱う必要があるのでかなり複雑です。

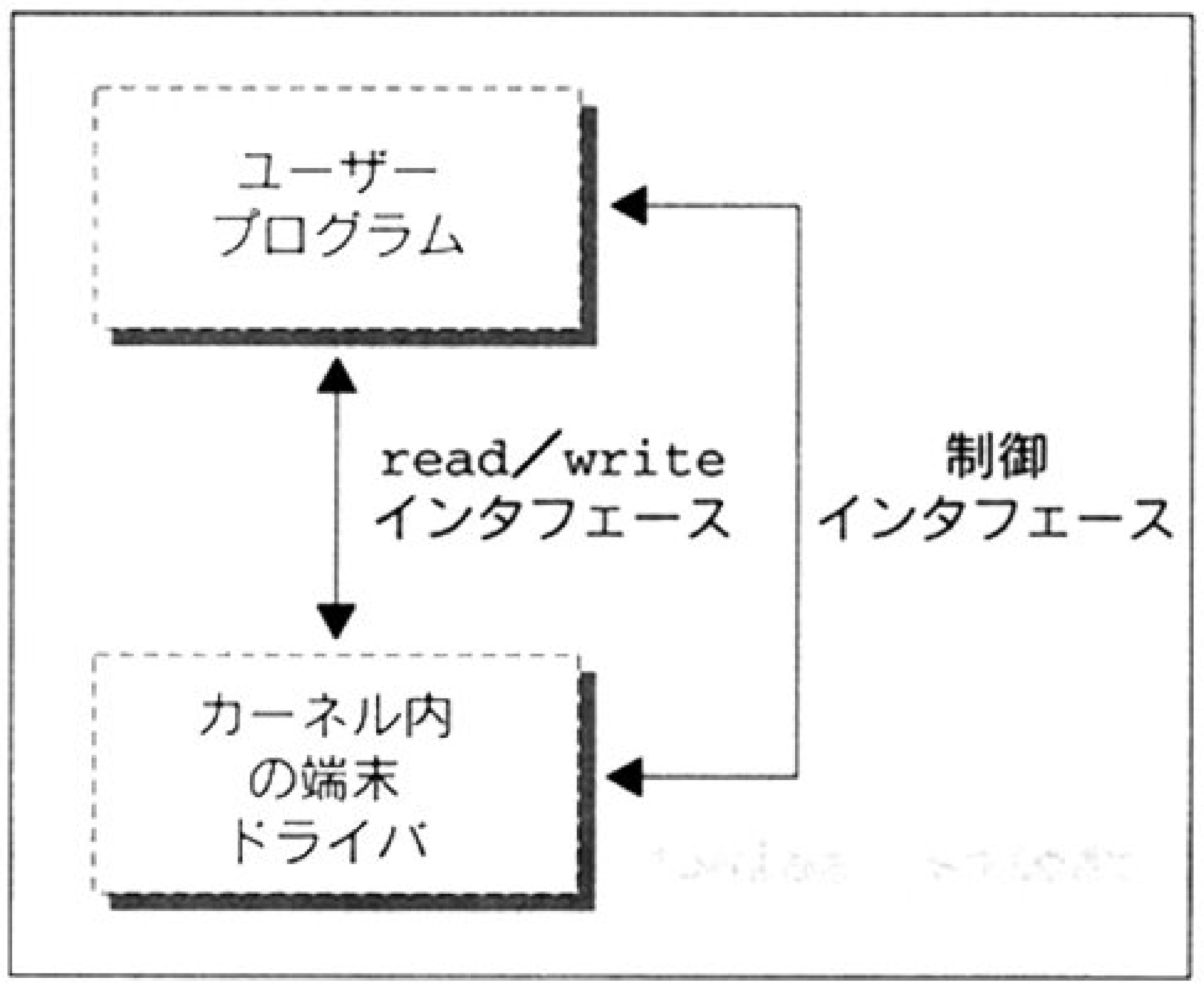


図5.1 汎用端末インタフェース

UNIX流のいい方をすれば、制御インタフェースは“ラインディシプリン”（回線制御手順）を設定します。このしくみによって、プログラムから端末ドライバの動作を柔軟に制御できるようになっています。

制御できる主な機能には、次のようなものがあります。

- 行編集　　**BackSpace**キーを使った編集の可否。
- バッファリング　　文字をすぐに読み取るか、それとも一定の間をおいてから読み取るかの指定。
- エコー　　パスワードを読み取る場合などのエコーの制御。
- **CR／LF**　　入力と出力のためのマッピング、**\n**の扱い。
- 回線速度　　PCのコンソールではほとんど使われない。ただし、モデムやシリアルラインを介して接続された端末では非常に重要。

5.2.1 ハードウェアモデル

汎用端末インタフェースについて詳しく取り上げる前に、このインタフェースが扱うハードウェアモデルについて説明しておきましょう。

ハードウェアモデルを図式的に表すと、次の図のようになります。UNIXマシンはシリアルポートを介してモデムと接続されており、さらにこのモデムから電話回線と別のモデムを経由してリモート端末と接続されています。これはあくまで概念的な構成を表すものですが、実際にこのような構成を持つUNIXサイトもあり、小規模なインターネットサービスプロバイダでは現実に図のような構成を使っていることがあります。この構成は、メインフレームでプログラムを実行し、ユーザーはダム端末で作業するという、かつてのクライアント／サーバー方式の遠い親戚ともいえます。

5

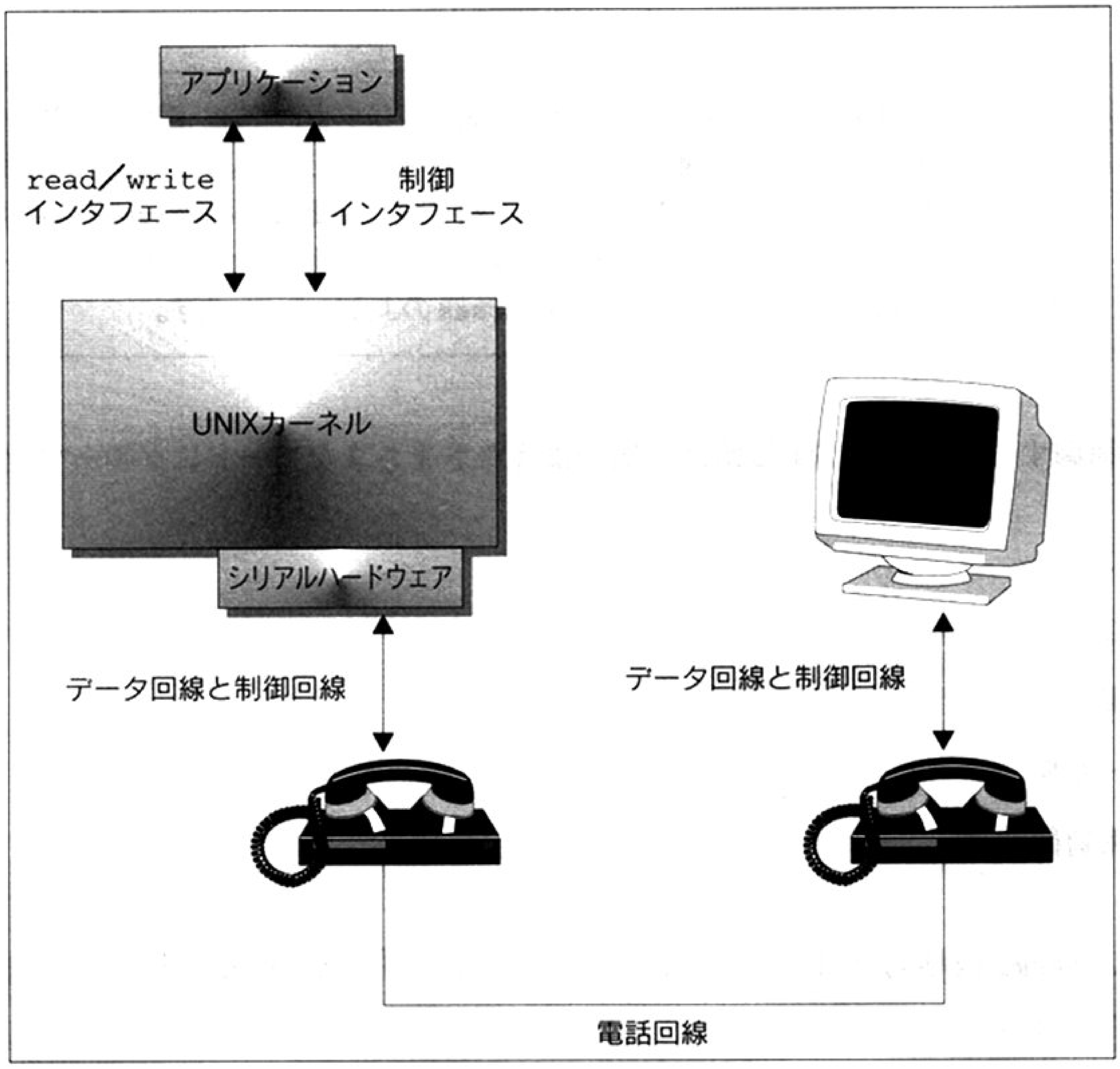


図5.2　汎用端末インタフェースのハードウェアモデル

PCでLinuxを動かしている場合、これは少し大げさなモデルに思えるかもしれません。しかし、筆者たちは二人ともマシンにモデムを接続しており、minicomなどの端末エミュレーションプログラムを使って互いに相手のマシンにリモートログインし、さまざまな操作を実行することができます。この場合、ちょうど上の図に示したように、一対のモデムと電話回線を使うことになります。

構成としては最も複雑なこのハードウェアモデルの利点は、現実世界のほとんどの状況をこのモデルのサブセットとして扱うことができる点にあります。つまり、このモデルより構成が単純であれば、それだけプログラムでのサポートも容易になります。

5.3 termios構造体

termiosはPOSIXで指定されている標準インタフェースで、System Vインタフェースの**termio**とも類似点があります。端末インタフェースを制御するには、**termios**型の構造体に値を設定し、関数呼び出しを使います。構造体と関数呼び出しはヘッダーファイル**termios.h**で定義されています。

> **注意**
> **termios.h**で定義されている関数呼び出しを使うプログラムでは、適切な関数ライブラリとリンクする必要があります。通常使うのは**curses**ライブラリです。たとえば、この章のサンプルプログラムをコンパイルするときには、コマンドラインの最後に**-lcurses**のように指定する必要があります。Linuxシステムでは**ncurses**をライブラリに使っていることも多く、この場合にはコンパイル時のコマンドラインで**-lncurses**のように指定します。

端末を制御するときに操作する値は、次のようなさまざまなモードにグループ化できます。

- 入力
- 出力
- 制御
- ローカル
- 特殊制御文字

一般に、**termios**構造体は次のように宣言されています（X/Open仕様では追加のフィールドも許容しています）。

```
#include <termios.h>

struct termios {
    tcflag_t c_iflag;
    tcflag_t c_oflag;
    tcflag_t c_cflag;
    tcflag_t c_lflag;
    cc_t     c_cc[NCCS];
};
```

termios構造体の各メンバには、上に示した5つのパラメータの種類に対応する名前が付いています。

tcgetattr関数を呼び出すと、端末に対応する**termios**構造体を初期化することができます。構文は次のとおりです。

```
#include <termios.h>

int tcgetattr(int fildes, struct termios, *termios_p);
```

tcgetattr関数は、端末インタフェース変数の現在の値を**termios_p**で示される構造体に書き込みます。端末インタフェース変数の値を変更したあと、**tcsetattr**関数を呼び出すと、端末インタフェースを設定し直すことができます。


```
#include <termios.h>

int tcsetattr(int fildes, int actions, const struct termios, *termios_p);
```

tcsetattr関数のパラメータ**actions**には、変更の適用方法として、次の3つのうちのいずれかを指定します。

- **TCSANOW**　値をただちに変更する。
- **TCSADRAIN**　現在の出力が完了したあとに値を変更する。
- **TCSAFLUSH**　現在の出力が完了したあとに値を変更するが、現在利用可能な入力のうち、**read**の呼び出しで返されていないものはすべて破棄する。

注意

> プログラムの中で端末の設定を変更する場合には、プログラムを終了するときに起動前の端末の設定を復元することが大切です。起動時の端末設定の保存と終了時の設定の復元は、常にプログラム側で責任を持って行わなければなりません。

以下では、各モードと、関連する関数呼び出しについて詳しく見ていきます。各モードには非常に特殊でほとんど使われない部分もあり、本書ではよく使われる主な機能だけを取り上げます。詳細を知りたい場合には、マニュアルページを読むか、またはPOSIXやX/Openの仕様を参照してください。

入力を読み取るときにまず考慮しなければならない最も重要なモードは、ローカルモードです。カノニカルモードと非カノニカルモードを使い分け、1行全体が入力されるまで待つか、それとも文字が入力されるごとに処理するかを指定すれば、この章で最初に作成したプログラムの2番目の問題も解決することができます。

5.3.1 入力モード

入力モードは、入力（シリアルポートまたはキーボードから端末ドライバが受け取った文字）をどのように処理してからプログラムに渡すかを制御します。具体的には、termios構造体のメンバc_iflagでフラグを設定します。フラグはいずれもマクロとして定義されており、これをビット単位の論理和で指定します。この指定方法は、入力モードだけでなく、すべての端末モードに適用されます。

c_iflagで使用できるマクロには次のものがあります。

- BRKINT　回線上でブレークを検出したときに割り込みを生成する。
- IGNBRK　回線上のブレークを無視する。
- ICRNL　受信したキャリッジリターンを改行に変換する。
- IGNCR　受信したキャリッジリターンを無視する。
- INLCR　受信した改行をキャリッジリターンに変換する。
- IGNPAR　パリティエラーの文字を無視する。
- INPCK　受信した文字に対してパリティチェックを行う。
- PARMRK　パリティエラーをマークする。
- ISTRIP　すべての入力文字の8ビット目を落とす。
- IXOFF　入力に対するソフトウェアフロー制御を有効にする。
- IXON　出力に対するソフトウェアフロー制御を有効にする。

注意

BRKINTとIGNBRKがどちらも設定されていない場合、回線上のブレークはNULL（0x00）文字として読み取られます。

通常、入力モードではデフォルト値が最も適切な値です。変更する必要はあまりありません。

5.3.2 出力モード

出力モードは、出力文字をどのように処理するか、つまりプログラムから送信された文字をどのように処理してからシリアルポートまたは画面に転送するかを制御します。出力モードのフラグの多くは、入力モードのフラグに対応しています。いくつか追加のフラグもありますが、これらは主にキャリッジリターンなどの文字の処理に時間がかかる低速の端末を対象とするフラグです。こうしたフラグの大部分は、現在では端末が高速化したために冗長になっているか、または

端末機能のデータベースであるterminfoを使うことでより適切に処理できます。terminfoについては、あとで取り上げます。

出力モードを指定するには、termios構造体のメンバc_oflagでフラグを設定します。c_oflagで使用できるマクロには次のものがあります。

- OPOST　出力処理を有効にする。
- ONLCR　出力される改行をキャリッジリターンとラインフィードのペアに変換する。
- OCRNL　出力されるキャリッジリターンを改行に変換する。
- ONOCR　0カラム目ではキャリッジリターンを出力しない。
- ONLRET　改行がキャリッジリターンも行う。
- OFILL　遅延に対して埋め込み文字を送信する。
- OFDEL　埋め込み文字としてNULLではなくDELを使う。
- NLDLY　改行遅延を選択する。
- CRDLY　キャリッジリターン遅延を選択する。
- TABDLY　タブ遅延を選択する。
- BSDLY　バックスペース遅延を選択する。
- VTDLY　垂直タブ遅延を選択する。
- FFDLY　フォームフィード遅延を選択する。

注意

> OPOSTが設定されていない場合、ほかのすべてのフラグは無視されます。

入力モード同様、出力モードもあまり使わないので、これ以上の説明はしません。

5.3.3 制御モード

制御モードは、端末のハードウェア特性を制御します。制御モードを指定するには、termios構造体のメンバc_cflagでフラグを設定します。c_cflagで使用できるマクロには次のものがあります。

- CLOCAL　モデムステータスラインを無視する。
- CREAD　文字の受信を可能にする。
- CS5　送受信文字に5ビットを使う。

- **CS6** 送受信文字に6ビットを使う。
- **CS7** 送受信文字に7ビットを使う。
- **CS8** 送受信文字に8ビットを使う。
- **CSTOPB** 1文字ごとに、1つではなく2つのストップビットを使う。
- **HUPCL** クローズ時にモデムをハングアップする。
- **PARENB** パリティの生成と検出を有効にする。
- **PARODD** 偶数パリティではなく奇数パリティを使う。

注意

> HUPCLが設定されている場合、端末を参照している最後のファイルデスクリプタがクローズされたことを検出すると、端末はモデム制御ラインを使って回線を切断（ハングアップ）します。

制御モードは、端末に対して使うこともありますが、主にシリアルラインにモデムが接続されている場合に使います。通常は、termiosの制御モードを使ってデフォルトの回線の動作を変更するより、端末自体の設定を変更するほうが容易です。

5.3.4 ローカルモード

ローカルモードは、端末のさまざまな特性を制御します。ローカルモードを指定するには、termios構造体のメンバc_lflagでフラグを設定します。c_lflagで使用できるマクロには次のものがあります。

- **ECHO** 入力文字のローカルエコーを有効にする。
- **ECHOE** ERASEの受信時に**BackSpace**、**Space**、**BackSpace**の組み合わせを実行する。
- **ECHOK** KILL文字で行の抹消を実行する。
- **ECHONL** 改行文字をエコーする。
- **ICANON** カノニカル入力処理を有効にする（下記参照）。
- **IEXTEN** 実装に依存する特殊機能を有効にする。
- **ISIG** シグナルを有効にする。
- **NOFLSH** キューのフラッシュを無効にする。
- **TOSTOP** 書き込みの試みに対してバックグラウンドプロセスにシグナルを送信する。

これらのフラグのうち、最も重要なのがECHOとICANONです。ECHOフラグを使うと、入力された文字がエコーされないようすることができます。ICANONフラグは、受信文字を処理する2つのモードを切り替えるときに使います。ICANONフラグが設定されている場合、回線はカノニカルモードになります。フラグが設定されていない場合には、非カノニカルモードになります。

カノニカルモードと非カノニカルモードについては、どちらのモードでも使う特殊制御文字について説明したあとで、詳しく取り上げます。

5.3.5 特殊制御文字

特殊制御文字は、ユーザーが入力したときに特別な方法で処理される**Ctrl - C**などの文字の集合です。termios構造体の配列c_ccには、特殊制御文字がサポートする各機能に割り当てられた文字が収められています。各文字の位置（配列の添字）は、マクロによって定義されていますが、各文字が制御文字でなければならないという制限はありません。

配列c_ccは、端末がカノニカルモードに設定されているかどうか（termios構造体のメンバc_lflagのICANONフラグの設定）に応じて、2つの異なる方法で使われます。

これら2つの異なるモードでの配列の添字は、部分的に重複していることに注意する必要があります。このため、2つのモードの値を混在させないようにしなければなりません。

カノニカルモードでの配列の添字には次のものがあります。

- **VEOF** EOF文字
- **VEOL** EOL文字
- **VERASE** ERASE文字
- **VINTR** INTR文字
- **VKILL** KILL文字
- **VQUIT** QUIT文字
- **VSUSP** SUSP文字
- **VSTART** START文字
- **VSTOP** STOP文字

非カノニカルモードでの配列の添字には次のものがあります。

- **VINTR** INTR文字
- **VMIN** MIN値
- **VQUIT** QUIT文字
- **VSUSP** SUSP文字

- **VTIME**　　TIME値
- **VSTART**　　START文字
- **VSTOP**　　STOP文字

特殊文字と非カノニカルモードのMINとTIMEの値は、入力文字を細かく制御するうえで非常に重要なので、少し詳しく取り上げます。

表5.1 特殊制御文字

文字	説明
INTR	この文字を受け取ると、端末ドライバは端末に接続されているプロセスに`SIGINT`シグナルを送る(シグナルの詳細については第10章を参照)。
QUIT	この文字を受け取ると、端末ドライバは端末に接続されているプロセスに`SIGQUIT`シグナルを送る。
ERASE	この文字を受け取ると、端末ドライバは回線上の最後の文字を削除する。
KILL	この文字を受け取ると、端末ドライバは行全体を削除する。
EOF	この文字を受け取ると、端末ドライバは入力を読み取っているアプリケーションに回線上のすべての文字を渡す。行が空の場合には、`read`を呼び出すと、ファイルの終わりに対する読み取り操作を行った場合と同様に0文字が返される。
EOL	この文字は、通常の改行文字に加えて行末記号として機能する。
SUSP	この文字を受け取ると、端末ドライバは端末に接続されているプロセスに`SIGSUSP`シグナルを送る。ジョブ制御をサポートしているUNIXの場合、現在のアプリケーションはサスペンド(停止)される。
STOP	この文字は、端末への出力を停止させる。XON/XOFFフロー制御をサポートするために使われ、一般にASCII XOFF文字**Ctrl-S**が設定される。
START	この文字は、STOP文字を受け取ったあとの出力を再開する。一般にASCII XON文字が使われる。

◆ TIMEとMINの値

TIMEとMINの値は非カノニカルモードでのみ使われ、いずれも入力の読み取りを制御する役割を持ちます。端末に関連付けられているファイルデスクリプタをプログラムで読み取る際の動作は、これらの値によって決まります。

以下の4つのケースがあります。

MIN = 0 かつ TIME = 0

`read`は、常にただちに戻ります。利用できる文字がある場合には、これらの文字が返されます。利用できる文字がない場合、`read`は0を返し、文字は読み取られません。

MIN = 0 かつ TIME > 0

`read`は、読み取る文字がある場合、または1/10秒を単位としてTIMEで指定された時間が経過した場合に戻ります。タイマが切れたために文字が読み取られなかった場合、`read`は0を返します。それ以外の場合には読み取った文字数を返します。

MIN > 0 かつ TIME = 0

readは、MINで指定された数の文字を読み取るまで待ち、その文字数を返します。ファイルの終わりに達した場合には0を返します。

MIN > 0 かつ TIME > 0

これは最も複雑なケースです。**read**は、1文字を受け取るまで待ちます。最初の文字を受け取ったとき、およびそれ以後1文字を受け取るごとに、文字間のタイマがスタートします（すでにスタートしている場合、タイマは再スタートします）。**read**は、MINで指定された数の文字を読み取るか、または1/10秒を単位としてTIMEで指定された文字間のタイマが切れると、呼び出しから戻ります。これは、**Esc**キーが1回押された場合と、ファンクションキーによるエスケープシーケンスの始まりとを区別する場合に便利です。ただし、ネットワーク経由で通信している場合やプロセッサの負荷が高い場合には、期待どおりの細かなタイミング情報を得られないことがあり、注意が必要です。

非カノニカルモードに設定してMINとTIMEの値を使えば、プログラムの中から入力に対する文字ごとの処理を行うことができます。

◆ シェルによる端末モードへのアクセス

シェルを使っているときに現在の**termios**の設定を知りたい場合には、次のコマンドを実行します。

```
$ stty -a
```

termios構造体が一部拡張されている筆者のLinuxシステムでは、次のように表示されます。

```
speed 38400 baud; rows 25; columns 80; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = <undef>;
eol2 = <undef>; start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W;
lnext = ^V; flush = ^O; min = 1; time = 0;
-parenb -parodd cs8 -hupcl -cstopb cread -clocal -crtscts
-ignbrk -brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl ixon ixoff
-iuclc -ixany -imaxbel
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0 vt0 ff0
isig icanon -iexten echo echoe echok -echonl -noflsh -xcase -tostop -echoprt
-echoctl echoke
```

この出力例では、EOF文字が**Ctrl - D**になっており、エコーが有効になっています。端末制御でいろいろな設定を試すと、端末が標準的ではない状態になり、端末の使用が困難になることがあります。このような状態になった場合には、いくつかの回避方法があります。

まず、**stty**が**sane**というオプションをサポートしている場合には、次のコマンドを使います。

```
$ stty sane
```

キャリッジリターンが改行文字（行の終わりを示す文字）に割り当てられていない状態になった場合には、**stty sane**と入力したうえで、**Return**キーではなく、**Ctrl-J**（改行文字）を入力する必要があります。

もうひとつの方法として、あらかじめ**stty -g**コマンドを使って、現在の**stty**の設定をあとで読み込める形で保存しておく方法があります。たとえば、次のようにします。

```
$ stty -g > save_stty
..
<端末の設定を変更>
..
$ stty $(cat save_stty)
```

この場合にも、最後に実行する**stty**コマンドでは、**Return**キーの代わりに**Ctrl-J**を入力する必要があるかもしれません。なお、上の方法はシェルスクリプトでも利用できます。

```
save_stty="$(stty -g)"
<端末の設定を変更>
stty $save_stty
```

もし上のような方法でも端末の設定を復元できない場合には、別の端末に切り替え、使用できなくなったシェルを**ps**コマンドで特定し、**kill HUP <process id>**を実行してシェルを終了させます。**stty**のパラメータはログインプロンプトが表示される前にリセットされるので、次回からまた正常にログインできるようになります。

◆ コマンドプロンプトからの端末モードの設定

sttyコマンドは、コマンドプロンプトから直接端末モードを設定する場合にも使用できます。

シェルスクリプトで1文字ごとの読み取りを行うようにモードを設定するには、カノニカルモードを無効にして、MINに1を、TIMEに0を設定します。具体的には、次のコマンドを実行します。

```
$ stty -icanon min 1 time 0
```

第2章でパスワードを読み取る簡単なシェルスクリプトを示しましたが、次のコマンドを実行すれば、エコーを無効にしてからパスワードの入力を求めることができます。

```
$ stty -echo
```

上のコマンドの実行後には、**stty echo**を使ってエコーを有効にしておく必要があります。

5.3.6 端末速度

termios構造体で提供される最後の機能は、回線速度の操作です。端末速度のためのメンバは定義されておらず、代わりに関数呼び出しを使って設定するようになっています。入力速度と出力速度はそれぞれ独立して扱われます。

全部で4つの関数があり、構文は次のとおりです。

```
#include <termios.h>

speed_t cfgetispeed(const struct termios *termios_p);
speed_t cfgetospeed(const struct termios *termios_p);
int cfsetispeed(struct termios *termios_p, speed_t speed);
int cfsetospeed(struct termios *termios_p, speed_t speed);
```


これらは、**termios**構造体に対して作用するもので、直接ポートに対して機能するわけではありません。つまり、新しい速度を設定するときは、まず**tcgetattr**を使って現在の設定を読み取り、上のいずれかの呼び出しを使って目的の速度を設定したあと、**tcsetattr**を使って**termios**構造体を書き戻す必要があります。回線速度は、**tcsetattr**の呼び出し後に初めて変更されます。

上の関数呼び出しの**speed**には、さまざまな値を指定できます。このうち主なものを次に示します。

- **B0**　端末をハングアップ
- **B1200**　1200ボー
- **B2400**　2400ボー
- **B9600**　9600ボー
- **B19200**　19200ボー
- **B38400**　38400ボー

38400より高い速度は、標準では定義されていません。また、これより高い速度でシリアルポートをサポートする標準的な方法もありません。

メモ

Linuxのバージョンによっては、**setserial**コマンドを使って非標準的な57600や115200の速度を設定できることもあります。これらの速度は、**B38400**が選択されている場合に使われます。ただし、この方法には移植性がないので、使用する場合には注意が必要です。

5.3.7 その他の関数

これまでにあげたもの以外にも、端末を制御するためのいくつかの関数があります。これらの関数は直接ファイルデスクリプタに対して使うもので、termios構造体の値の取得や設定は必要ありません。構文は次のとおりです。

```
#include <termios.h>

int tcdrain(int fildes);
int tcflow(int fildes, int flowtype);
int tcflush(int fildes, int in_out_selector);
```

これらの関数は、次のような目的で使います。

- tcdrain関数は、キューに入っているすべての出力の送信が完了するまで、呼び出し元のプログラムを待機させます。
- tcflow関数は、出力を停止したり再開したりする場合に使います。
- tcflush関数は、入力または出力、あるいはその両方をフラッシュする場合に使います。

以上で、termios構造体についての説明は終わりです。以下では、具体的な例を使ってtermios構造体の使い方を学びます。まず、最も単純な例として、エコーを無効にしてパスワードを読み取るプログラムを作成してみましょう。この場合には、ECHOフラグをオフにします。

例題 termiosを使ったパスワードプログラム

1 作成するプログラムの名前はpassword.cとします。最初に、次のような定義を記述します。

```
#include <termios.h>
#include <stdio.h>

#define PASSWORD_LEN 8

int main()
{
    struct termios initialrsettings, newrsettings;
    char password[PASSWORD_LEN + 1];
```

2 標準出力の現在の設定を取得し、用意したtermios構造体に設定をコピーします。

```
    tcgetattr(fileno(stdin), &initialrsettings);
```

3 もとの設定のコピーを作成し、これを使って設定を変更します。newrsettingsのECHOフラグをオフにして、ユーザーにパスワードの入力を求めます。

```
    newrsettings = initialrsettings;
    newrsettings.c_lflag &= ~ECHO;

    printf("Enter password: ");
```

4 次に、端末の属性をnewrsettingsに設定し、パスワードを読み取ります。最後に、端末の属性をもとに戻し、入力されたパスワードを表示します。

```
    if(tcsetattr(fileno(stdin), TCSAFLUSH, &newrsettings) != 0) {
        fprintf(stderr,"Could not set attributes\n");
    }
    else {
        fgets(password, PASSWORD_LEN, stdin);
        tcsetattr(fileno(stdin), TCSANOW, &initialrsettings);
        fprintf(stdout, "\nYou entered %s\n", password);
    }
    exit(0);
}
```


プログラムを実行すると、次のように出力されます。

```
$ password
Enter password:
You entered hello

$
```

解説

実行例では、Enter password:のプロンプトに対してhelloと入力していますが、入力した文字のエコーは行われず、ユーザーが**Return**キーを押すまで画面にはなにも出力されません。

プログラムでは、X &= ~FLAGという構文を使って、変更する必要のあるフラグだけを変更しています。この構文は、変数XのうちFLAGで定義されているビットをクリアします。同じような構文を使ってX |= FLAGとすれば、FLAGで定義されたビットだけをセットすることができます。

属性を設定する際、TCSAFLUSHを使って、すでに入力された文字があればそれを破棄しています。これは、エコーが無効にならないうちは、ユーザーにパスワードを入力させないようにするための設定です。端末の設定は、プログラムを終了する前にもとに戻します。

さて、termios構造体のもうひとつの一般的な使い方として、端末の設定を変更して入力と同時に文字を読み取ることができるようにするケースがあります。このようなケースでは、カノニカルモードを無効にしてMINとTIMEを適切に設定します。

例題 文字単位での読み取り

1 これまでに学んだ知識を応用すれば、この章の最初で作成したメニュープログラムを改善することができます。具体的には、`password.c`に似たコードを`menu3.c`に追加します。新しいプログラムの名前は`menu4.c`とします。まず、プログラムの最初で必要なファイルをインクルードします。

```
#include <stdio.h>
#include <unistd.h>
#include <termios.h>
```

2 `main`関数でいくつかの新しい変数を宣言します。

```
    int choice = 0;
    FILE *input;
    FILE *output;
    struct termios initial_settings, new_settings;
```

3 `getchoice`関数を呼び出す前に、端末の属性を変更しておきます。

```
        fprintf(stderr, "Unable to open /dev/tty\n");
        exit(1);
    }
    tcgetattr(fileno(input),&initial_settings);
    new_settings = initial_settings;
    new_settings.c_lflag &= ~ICANON;
    new_settings.c_lflag &= ~ECHO;
    new_settings.c_cc[VMIN] = 1;
    new_settings.c_cc[VTIME] = 0;
    new_settings.c_lflag &= ~ISIG;
    if(tcsetattr(fileno(input), TCSANOW, &new_settings) != 0) {
        fprintf(stderr,"could not set attributes\n");
    }
```

4 プログラムを終了する前に、設定をもとに戻します。

```
    do {
        choice = getchoice("Please select an action", menu, input, output);
        printf("You have chosen: %c\n", choice);
    } while (choice != 'q');
    tcsetattr(fileno(input),TCSANOW,&initial_settings);
    exit(0);
}
```

5 今度は非カノニカルモードになっているので、CRからLFへのデフォルトマッピングは行われません。このため、キャリッジリターンについてもチェックする必要があります。

```
    do {
        selected = fgetc(in);
    } while (selected == '\n' || selected == '\r');
```

6 新しいプログラムは、ユーザーが**Ctrl - C**を入力した時点で終了してしまいます。このため、ローカルモードの`ISIG`フラグをクリアして特殊文字の処理を無効にします。具体的には、次の行を`main`関数に追加します。

```
new_settings.c_lflag &= ~ISIG;
```


解説

変更を加えた新しいプログラムを実行すると、入力と同時に応答が返ります。ただし、入力した文字はエコーされません。

```
$ menu4
Choice: Please select an action
a - add new record
d - delete record
q - quit
You have chosen: a
Choice: Please select an action
a - add new record
d - delete record
q - quit
You have chosen: q
$
```

Ctrl - Cを入力すると、プログラムに直接入力が渡され、不適切な選択肢として処理されます。

5.4 端末出力

`termios`構造体を使うとキーボード入力を制御できますが、プログラムからの出力を画面に表示する方法も同じようなレベルで制御できると便利です。この章の最初に作成したプログラムでは、`printf`を使って画面に文字を出力しましたが、画面上の特定の位置に文字を出力する手段はありませんでした。

◆ 端末の種類

多くのUNIXシステムは端末と一緒に使います。今日では、一般に端末プログラムを実行しているPCが端末としての機能を果たしています。しかし、歴史的には、さまざまなメーカーから多数の端末が提供されてきた経緯があります。これらの端末の大部分は、エスケープシーケンス（エスケープ文字で始まる文字列）を使って、カーソルの位置や、ボールド、点滅といったその他の属性を制御していますが、その方法は十分に標準化されていません。一部の古い端末ではスクロール機能が互いに異なっていたり、バックスペースが送られたときに文字を消すものやそうでないものなどがあります。

メモ

エスケープシーケンスには、ANSI標準のものがあります。これは、主にDECのVTシリーズ端末で使われていたエスケープシーケンスに基づいていますが、完全に同じではありません。多くのPC端末プログラムは、VT100、VT220、ANSIなどの標準端末に加えて、その他の端末のエミュレーション機能も持っています。

端末の種類が多い点は、画面制御を行うプログラムをさまざまな端末で動かしたいと考えるプログラマにとって非常にやっかいな問題です。たとえば、ANSI端末ではカーソルを1行上に移動するのに**Esc-[-A**というエスケープシーケンスを使いますが、数年前までよく使われていたADM-3a端末では**Ctrl-K**を使います。

このため、UNIXシステムに接続されるさまざまな種類の端末で使用できるプログラムを作成しようとすると、端末の種類ごとに異なるソースコードを用意する必要があり、プログラマにとって大きな負担になってしまいます。

さいわいなことに、こうした煩雑な作業を避ける方法として`terminfo`というパッケージが用意されています。`terminfo`では、各プログラムで端末ごとの違いを処理する代わりに、端末の種類を網羅したデータベースをプログラムから参照し、必要な情報を取得できるようになっています。`terminfo`は、ほとんどの新しいUNIXシステムでは`curses`と呼ばれるもうひとつのパッケージと組み合わせて使います。`curses`については、次の章で詳しく取り上げます。

Linuxの場合には、`curses`の実装である`ncurses`を使用し、`terminfo`関連関数のプロトタイプを提供するために`ncurses.h`（ディストリビューションによっては`curses.h`）をインクルードします。`terminfo`関連関数自体は、独自のヘッダーファイル`term.h`で宣言されています。（少なくともこれまではそうでした。Linuxの新しいバージョンでは、`terminfo`と`ncurses`の境界があいまいになっており、`terminfo`関連関数を使うプログラムで`ncurses`ヘッダーファイルもインクルードしなければならないことがあります）。

5.4.1 端末の種類の特定

UNIXには`TERM`という環境変数があり、この変数に、現在使われている端末の種類が設定されます。通常、`TERM`は、ユーザーのログオン時にシステムによって自動的に設定されます。システム管理者は、直接接続されている各端末についてはデフォルトの端末の種類を設定し、ネットワーク経由でアクセスする場合にはユーザーが端末の種類を指定できるようにしていることがあります。`TERM`の値は、`telnet`を介して設定でき、また`rlogin`では、リモートログインする前の端末の値がそのまま渡されます。

現在使っている端末をシステムがどのように認識しているかは、次のコマンドで知ることができます。


```
$ echo $TERM
xterm
$
```

この例では、`xterm`と呼ばれるプログラムからシェルが実行されています。`xterm`はX Window System用の端末エミュレータです。

`terminfo`パッケージは、多数の端末の機能とエスケープシーケンスのデータベースを含んでおり、これらを使うための統一されたプログラミングインタフェースを提供しています。`terminfo`パッケージを使えば、1つのプログラムを書くだけですみ、アプリケーション側で多くの異なる端末をサポートする必要はありません。また、将来新しい端末が登場しても、`terminfo`データベースが更新されれば、それだけで新しい端末を使用できるようになります。

`terminfo`の機能は属性ごとに記述され、コンパイル済みの一連の`terminfo`ファイルに格納されています。これらのファイルは、一般に`/usr/lib/terminfo`に置かれます。端末（および`terminfo`でも指定可能なプリンタ）ごとに、その機能と使い方を定義したファイルが存在します。実際のファイルは、ディレクトリが大きくなるのを避けるために、端末の種類の先頭の1文字を名前に持つサブディレクトリに置かれています。たとえば、VT100の定義は`/usr/lib/terminfo/v/vt100`にあります。

`terminfo`ファイルは端末の種類ごとに1つ作成されます。人間が読めるソース形式のものを`tic`コマンドでコンパイルし、アプリケーションプログラムから利用しやすいコンパクトで効率的な形式に変換するようになっています。X/Open仕様では、ソースの形式とコンパイル後の形式については定義がありますが、奇妙なことに実際にソースからコンパイル済み形式に変換するための`tic`コマンドについては言及がありません。

次に示すのは、VT100端末の`terminfo`ファイルの例です。

```
vt100|vt100-am|dec vt100 (w/advanced video),
 am, msgr, xenl, xon,
 cols#80, it#8, lines#24, vt#3,
 acsc=``aaffggjjkkllmmnnooppqqrrssttuuvvwwxxyyzz{{||}}~~,
 bel=^G, blink=\E[5m$<2>, bold=\E[1m$<2>,
 clear=\E[H\E[J$<50>, cr=^M, csr=\E[%i%p1%d;%p2%dr,
 cub=\E[%p1%dD, cub1=^H, cud=\E[%p1%dB, cud1=^J,
 cuf=\E[%p1%dC, cuf1=\E[C$<2>,
 cup=\E[%i%p1%d;%p2%dH$<5>, cuu=\E[%p1%dA,
 cuu1=\E[A$<2>, ed=\E[J$<50>, el=\E[K$<3>,
 el1=\E[1K$<3>, enacs=\E(B\E)0, home=\E[H, ht=^I,
 hts=\EH, ind=^J, ka1=\EOq, ka3=\EOs, kb2=\EOr,
 kbs=^H, kc1=\EOp, kc3=\EOn, kcub1=\EOD, kcud1=\EOB,
 kcuf1=\EOC, kcuu1=\EOA, kent=\EOM, kf0=\EOy,
 kf1=\EOP, kf10=\EOx, kf2=\EOQ, kf3=\EOR, kf4=\EOS,
 kf5=\EOt, kf6=\EOu, kf7=\EOv, kf8=\EOl, kf9=\EOw,
 rc=\E8, rev=\E[7m$<2>, ri=\EM$<5>, rmacs=^O,
 rmam=\E[?7l, rmkx=\E[?1l\E>, rmso=\E[m$<2>,
 rmul=\E[m$<2>,
 rs2=\E>\E[?3l\E[?4l\E[?5l\E[?7h\E[?8h, sc=\E7,
 sgr=\E[0%?%p1%p6%|%t;1%;%?%p2%t;4%;%?%p1%p3%|%t;7%;%?%p4%t;5%;m%?%p9%t\016%e\017%;,
 sgr0=\E[m\017$<2>, smacs=^N, smam=\E[?7h,
 smkx=\E[?1h\E=, smso=\E[7m$<2>, smul=\E[4m$<2>,
 tbc=\E[3g,
```

terminfoの定義は、3つの種類のエントリから構成されています。各エントリはcapnameと呼ばれ、端末の機能を定義しています。

ブール型機能は、該当する機能を端末がサポートしているかどうかを示します。たとえば、ブール型機能xonが存在する場合、端末はXON/XOFFフロー制御をサポートしています。また、cub1が存在する場合には、0カラム目にあるカーソルを1つ左に移動すると、上の行の最後に移動します。

数値型機能はサイズを定義します。たとえば、linesは画面の行数、colsは画面のカラム数を表します。機能名と実際の数値の間には#文字を入れます。たとえば、80カラムで24行の端末を定義するには、cols#80, lines#24と記述します。

文字列型機能は上の2つの機能と比べてやや複雑で、2つの異なる目的で使われます。ひとつは、端末機能にアクセスするのに必要な出力文字列を定義するためで、もうひとつは、主にファンクションキーやテンキー上の特殊キーなどの特定のキーをユーザーが押したときに受け取る入力文字列を定義するためです。文字列型機能の中には、行末までを削除するelのように、非常にシンプルなものもあります。たとえば、VT100端末上で行末までを削除するエスケープシーケンスは**Esc**-**[**-**K**です。これをterminfoのソース形式で表すと、el=\E[Kになります。

特殊キーも同様の方法で定義されます。たとえば、VT100のファンクションキー**F1**は、**Esc**-**O**-**P**というエスケープシーケンスを送出します。これはkf1=\EOPと定義されます。

事情が込み入ってくるのは、いくつかのパラメータを必要とするエスケープシーケンスの場合です。ほとんどの端末は、指定された行とカラムにカーソルを移動することができます。しかし、

カーソルの位置ごとに個別に機能を用意することは現実には不可能です。このため、パラメータを使う汎用的な機能文字列を用意し、実際に文字列が使われるときにパラメータを値で置き換えるようになっています。

たとえば、VT100端末は、**Esc - [- <row> - ; - <col> - H**というエスケープシーケンスを使ってカーソルを特定の位置に移動します。これを`terminfo`のソース形式で表すと、`cup=\E[%i%p1%d;%p2%dH$<5>`になります。

各要素の意味は次のとおりです。

- `\E`　エスケープを送出する。
- `[`　文字`[`を送出する。
- `%i`　引数をインクリメントする。
- `%p1`　最初の引数をスタックに積む。
- `%d`　スタックの数値を10進数として出力する。
- `;`　文字`;`を送出する。
- `%p2`　2番目の引数をスタックに積む。
- `\d`　スタックの数値を10進数として出力する。
- `H`　文字`H`を送出する。

かなり複雑な文字列ですが、この記法を使うことで、端末が期待する最終的なエスケープシーケンス内での各パラメータの順序とは関係なく、固定した順序でパラメータを並べることができます。上の例で、引数をインクリメントする`%i`が使われているのは、標準的なカーソル指定では画面左上の起点を(0,0)としているのに対して、VT100では(1,1)を起点としているためです。最後の`$<5>`は、端末がカーソル移動を処理するのに、5文字を出力する時間に等しいだけの遅延が必要であることを示します。

メモ

端末の機能については非常に多くのものを定義できますが、ほとんどのUNIXシステムでは大部分の端末があらかじめ定義されています。新しい端末を追加する必要がある場合には、`terminfo`のマニュアルに記載されている機能のリストが参考になります。また、新しい端末に似た端末があれば、その端末をひな型として使うと便利です。自分で機能を定義するときには、必要に応じてデータベースを更新しながら、1つずつ項目を追加していくとよいでしょう。

マニュアルページ以外の標準的なリファレンスとしては、O'Reilly&Associates刊『Termcap and Terminfo』(邦訳『termcap & terminfo』アスキー刊)があります。

5.4.2 terminfo機能の使用

端末の機能の定義方法は上に説明したとおりですが、機能にアクセスする方法についても知っておく必要があります。terminfoを使う場合には、まず最初にsetuptermを呼び出して端末の種類をセットアップする必要があります。setuptermを呼び出すと、現在の端末の種類のTERMINAL構造体が初期化されます。この時点で、端末の機能を問い合わせたり、実際に機能を使用したりすることが可能になります。setuptermの構文は次のとおりです。

```
#include <term.h>

int setupterm(char *term, int fildes, int *errret);
```

setuptermライブラリ関数は、termで指定された端末の種類を現在の端末の種類として設定します。termがヌルポインタの場合には、環境変数TERMの値が使われます。端末への書き込みに使われるオープンされたファイルデスクリプタは、パラメータfildesで渡す必要があります。呼び出しの結果は、errretで示されるint型変数に格納されます（errretがヌルポインタではない場合）。書き込まれる値は次のいずれかです。

- -1　terminfoデータベースが存在しない
- 0　一致するエントリがterminfoデータベースにない
- 1　成功

setupterm関数は、成功すると定数OKを返し、失敗した場合にはERRを返します。errretがヌルポインタの場合には、失敗すると診断メッセージを表示してプログラムを終了します。次に例を示します。

```
#include <stdio.h>
#include <term.h>
#include <ncurses.h>

int main()
{
    setupterm("unlisted", fileno(stdout), (int *)0);
    printf("Done.\n");
    exit(0);
}
```

このプログラムの実行例を次に示します。表示されるメッセージはシステムによって多少異なるかもしれませんが、意味は同じです。結果を見ればわかるとおり、Done.は出力されません。これは、setuptermが失敗した時点でプログラムが終了するからです。

```
$ gcc -o badterm badterm.c -I/usr/include/ncurses -lncurses
$ badterm
couldn't open terminfo file /usr/lib/terminfo/u/unlisted
The terminal you are using is not defined.
$
```

上のコンパイルを実行している行に注目してください。筆者のLinuxシステムでは、ヘッダーファイル`ncurses.h`は`/usr/include/ncurses`ディレクトリにあります。このため、`-I`オプションを使って、インクルードファイルを検索するディレクトリを明示的に指定しています。

メニュー選択プログラムでは、画面をクリアしたり、カーソルを画面の指定した位置に移動してメッセージを表示したりできると便利です。`setupterm`を呼び出したあとは、端末機能の種類に対応した次の3つの関数呼び出しを使って、`terminfo`の機能にアクセスすることができます。


```
#include <term.h>

int tigetflag(char *capname);
int tigetnum(char *capname);
char *tigetstr(char *capname);
```

`tigetflag`、`tigetnum`、`tigetstr`は、それぞれブール型、数値型、文字列型の`terminfo`の機能の値を返します。指定された機能が存在しないなどの理由で呼び出しが失敗した場合、`tigetflag`は-1、`tigetnum`は-2、`tigetstr`は`(char *)-1`を返します。

では、`terminfo`データベースを使って`cols`機能と`lines`機能の値を取得し、端末のサイズを調べてみましょう。プログラムの名前は`sizeterm.c`とします。

```
#include <stdio.h>
#include <term.h>
#include <ncurses.h>

int main()
{
    int nrows, ncolumns;

    setupterm(NULL, fileno(stdout), (int *)0);
    nrows = tigetnum("lines");
    ncolumns = tigetnum("cols");
    printf("This terminal has %d columns and %d rows\n", ncolumns, nrows);
    exit(0);
}
```

プログラムの実行例を次に示します。

```
$ echo $TERM
vt100
$ sizeterm
This terminal has 80 columns and 24 rows
$
```

このプログラムをウィンドウの中で実行すると、次のように現在のウィンドウのサイズが表示されます。

```
$ echo $TERM
xterm
$ sizeterm
This terminal has 88 columns and 40 rows
$
```

tigetstrを使って**xterm**のカーソル移動機能（**cup**）を取得すると、**\E[%p1%d;%p2%dH**が返されます。

この機能を使うには、カーソル移動先の行とカラムに対応する2つのパラメータが必要です。これらの座標は、画面の左上隅を0として指定します。

機能に含まれるパラメータを実際の値に置き換えるには、**tparm**関数を使います。パラメータは9個まで置換でき、出力可能なエスケープシーケンスが返されます。

```
#include <term.h>

char *tparm(char *cap, long p1, long p2, ..., long p9);
```

◆ 端末への制御文字列の出力

tparmで作成したエスケープシーケンスは、端末に送らなければなりません。この処理を正しく行うためには、**printf**を使って端末に文字列を送るのではなく、端末が操作を完了するのに必要な遅延を処理できる特別な関数、**putp**と**tputs**を使います。

```
#include <term.h>

int putp(char *const str);
int tputs(char *const str, int affcnt, int (*putfunc)(int));
```

putp関数は、成功すると**OK**を返し、失敗すると**ERR**を返します。**putp**は端末制御文字を受け取り、これを**stdout**に送ります。古いLinuxディストリビューションの中には、**tputs**関数の最後のパラメータを**int (*putfunc)(char)**として定義しているものがあります。このようなディストリビューションを使っている場合には、あとで示すサンプルプログラムの**char_to_terminal**関数の定義を変更する必要があります。

たとえば、画面5行目の30カラム目に移動するには、次のようにコードを記述します。

```
char *cursor;
char *esc_sequence;
cursor = tigetstr("cup");
esc_sequence = tparm(cursor,5,30);
putp(esc_sequence);
```

tputs関数は、端末がstdout経由でアクセスされていない場合に、文字出力用の関数を指定できるようにするために用意されています。tputsは、成功するとOKを返し、失敗するとERRを返します。パラメータaffcntは、変更によって影響を受ける行数を指定するのに使います。通常は1を指定します。文字列を出力するのに使う関数は、パラメータと戻り値の型がputchar関数と同じでなければなりません。実際、putp(string)は、tpus(string, 1, putchar)を呼び出すのと等価です。ユーザー指定の出力関数を指定したtputsの使い方についてはあとで示します。

メモ

tparmや端末機能についてのマニュアルページを参照すると、tgotoという関数があることに気づくかもしれません。この関数を使えばもっと簡単にカーソルを移動できそうですが、本書ではtgotoを使っていません。これは、本書執筆時点でのX/Open仕様（1995年1月のバージョン）が、tgetent、tgetflag、tgetnum、tgetstr、tgotoの各関数の「使用をやめるべき」としているからです。読者がこれから作成するプログラムでも、これらの関数は使わないほうがよいでしょう。

さて、これまでに説明したことがらをふまえて、メニュー選択プログラムに画面制御機能を追加してみましょう。画面をクリアする方法も必要になりますが、これにはclearという機能を利用できます。端末によってはclearをサポートしていないことがありますが、この場合にはカーソルを画面左上隅に移動し、カーソルよりあとの表示をすべて消去するedを使えば目的を達成できます。

次に示すのは、メニュー選択プログラムの最終バージョン、screenmenu.cです。画面にオプションを表示し、有効なオプションの選択をユーザーに求めます。

例題 完全な端末制御

menu4.cのgetchoice関数に修正を加え、画面制御機能を追加します。main関数は同じなので、次に示すコードから省いています。menu4.cと違う部分については、わかりやすくするために強調表示しています。

```
#include <stdio.h>
#include <unistd.h>
#include <termios.h>
#include <term.h>
#include <curses.h>

static FILE *output_stream = (FILE *)0;
```

```
char *menu[] = {
    "a - add new record",
    "d - delete record",
    "q - quit",
    NULL,
};

int getchoice(char *greet, char *choices[], FILE *in, FILE *out);
int char_to_terminal(char char_to_write);

int main()
{
...
}

int getchoice(char *greet, char *choices[], FILE *in, FILE *out)
{
    int chosen = 0;
    int selected;
    int screenrow, screencol = 10;

    char **option;
    char *cursor, *clear;

    output_stream = out;

    setupterm(NULL,fileno(out), (int *)0);
    cursor = tigetstr("cup");
    clear = tigetstr("clear");

    screenrow = 4;
    tputs(clear, 1, char_to_terminal);
    tputs(tparm(cursor, screenrow, screencol), 1, char_to_terminal);
    fprintf(out, "Choice: %s", greet);
    screenrow += 2;
    option = choices;
    while(*option) {
        tputs(tparm(cursor, screenrow, screencol), 1, char_to_terminal);
        fprintf(out,"%s", *option);
        screenrow++;
        option++;
    }

    do {
        selected = fgetc(in);
        option = choices;
        while(*option) {
            if(selected == *option[0]) {
                chosen = 1;
                break;
            }
            option++;
```

```
        }
        if(!chosen) {
            tputs(tparm(cursor, screenrow, screencol), 1, char_to_terminal);
            fprintf(out,"Incorrect choice, select again\n");
        }
    } while(!chosen);
    tputs(clear, 1, char_to_terminal);
    return selected;
}

int char_to_terminal(char char_to_write)
{
    if (output_stream) putc(char_to_write, output_stream);
    return 0;
}
```


解説

修正後の**getchoice**関数では以前と同じメニューを使っていますが、表示ルーチンを変更し、**terminfo**の機能を利用しています。**You have chosen:**というメッセージも表示されますが、すぐに画面がクリアされてしまうためによく見えないかもしれません。この場合には、**main**関数で次のように**sleep**を呼び出すと、メッセージをはっきりと確認することができます。

```
do {
    choice = getchoice("Please select an action", menu, input, output);
    printf("You have chosen: %c\n", choice);
    sleep(1);
} while (choice != 'q');
```

プログラムの最後の関数**char_to_terminal**では、第3章で説明した**putc**関数を使っています。

さて、この章を終える前に、キーストロークを検出する方法について簡単な例を使って説明しておきましょう。

5.5 キーストロークの検出

MS-DOSでプログラムを作成した経験がある場合には、キーが押されたかどうかを返す**kbhit**に相当する関数がUNIXにもあれば、と思うかもしれません。しかし、残念ながら同等の関数は存在しません。UNIXプログラマは、**kbhit**に相当する関数がなくても困ることはありません。UNIXでは、通常、あるイベントをずっと待ち続けるような形でプログラムを作成することはないからです。**kbhit**はまさにこうした目的のために使う関数なので、UNIXプログラマがこの関

数の必要性を感じることはまずありません。

ただし、MS-DOSからプログラムを移植する場合には、kbhit関数をエミュレートする方法があると便利です。次に示すのは、非カノニカルモードを使ってkbhit相当の機能を実現する例です。

例題 独自のkbhit

1 プログラムの最初で、端末設定に必要なヘッダーファイルと構造体について記述します。peek_characterは、キーが押されたかどうかのテストに使います。プログラムの中で用意した関数についてもプロトタイプを宣言しておきます。

```
#include <stdio.h>
#include <termios.h>
#include <term.h>
#include <curses.h>
#include <unistd.h>

static struct termios initial_settings, new_settings;
static int peek_character = -1;

void init_keyboard();
void close_keyboard();
int kbhit();
int readch();
```

2 main関数では、init_keyboardを呼び出して端末を適切に設定したあと、1秒に1回ループを回りながらkbhitを呼び出します。入力されたキーがqの場合には、close_keyboardを呼び出して端末の設定をもとに戻し、プログラムを終了します。

```
int main()
{
    int ch = 0;

    init_keyboard();
    while(ch != 'q') {
        printf("looping\n");
        sleep(1);
        if(kbhit()) {
            ch = readch();
            printf("you hit %c\n",ch);
        }
    }
    close_keyboard();
    exit(0);
}
```

3 init_keyboardとclose_keyboardは、それぞれプログラムの起動時と終了時に端末の設定を行います。

```
void init_keyboard()
{
    tcgetattr(0,&initial_settings);
    new_settings = initial_settings;
    new_settings.c_lflag &= ~ICANON;
    new_settings.c_lflag &= ~ECHO;
    new_settings.c_lflag &= ~ISIG;
    new_settings.c_cc[VMIN] = 1;
    new_settings.c_cc[VTIME] = 0;
    tcsetattr(0, TCSANOW, &new_settings);
}

void close_keyboard()
{
    tcsetattr(0, TCSANOW, &initial_settings);
}
```

4 kbhitは、キーが押されたかどうかをチェックします。

```
int kbhit()
{
    char ch;
    int nread;

    if(peek_character != -1)
        return 1;
    new_settings.c_cc[VMIN]=0;
    tcsetattr(0, TCSANOW, &new_settings);
    nread = read(0,&ch,1);
    new_settings.c_cc[VMIN]=1;
    tcsetattr(0, TCSANOW, &new_settings);

    if(nread == 1) {
        peek_character = ch;
        return 1;
    }
    return 0;
}
```

5 押された文字は、readch関数で読み取ります。readch関数では、次のループに備えるためにpeek_characterに-1を設定します。

```
int readch()
{
    char ch;

    if(peek_character != -1) {
```

```
        ch = peek_character;
        peek_character = -1;
        return ch;
    }
    read(0,&ch,1);
    return ch;
}
```

プログラムを実行すると、次のように出力されます。

```
$ kbhit
looping
looping
looping
you hit h
looping
looping
you hit d
looping
you hit q
$
```

解説

init_keyboardでは、1文字を読み取ってから戻るように端末を設定しています（MIN=1、TIME=0）。kbhitではこの設定をさらに変更し、入力をチェックしてすぐに戻るようにして（MIN=0、TIME=0）、関数を抜ける前にもとの設定に戻しています。

押されたキーを読み込む必要がありますが、プログラムではこれをローカルに格納し、要求されたときに返せるようにしています。

5.6 この章のまとめ

この章では、端末の制御に関するさまざまな側面について取り上げました。最初の部分では、リダイレクトを検出する方法、標準ファイルデスクリプタがリダイレクトされている場合に端末と直接やりとりする方法について説明しました。

次に、汎用端末インタフェースと、UNIX端末の扱い方を細かく制御するためのtermios構造体について解説しました。

最後に、terminfoデータベースと関連関数の使い方を示し、端末に依存しない形で画面出力を制御する方法を学びました。

第6章

curses

第5章では、文字の入力を細かく制御する方法と、端末に依存しない形で文字を出力する方法について学びました。

しかし、汎用端末インタフェース（termios）を利用し、tparmや関連関数を使ってエスケープシーケンスを操作するには、かなりの量の低水準コードを記述する必要があります。もし、より高水準のインタフェースが利用可能ならば、そのほうがプログラムの作成も容易になります。たとえば、単に画面に出力するだけで、あとはライブラリ関数が端末に依存する部分を自動的に処理してくれるインタフェースがあれば便利です。

この章では、そうしたライブラリのひとつであるcursesを取り上げます。cursesライブラリを使うと、単純な行ベースのプログラムと、完全にグラフィカルな（ただしプログラミングもそれだけ困難な）X Window上のプログラムとのちょうど中間レベルのプログラムを作成できます。cursesライブラリでは、文字ベースながらも端末に依存せずにフルスクリーンで動作するプログラムを比較的簡単に作成できるので、多くのプログラムがcursesを使っています。文字ベースで画面指向のプログラムを作成する場合には、一般にエスケープシーケンスを直接書き込むよりcursesを使ったほうがはるかに楽です。cursesはキーボードも管理できるので、扱いの容易なブロックしない文字入力モードを利用することもできます。テキストエディタのviでも、cursesライブラリを使っています。

この章で取り上げる項目は次のとおりです。

- cursesライブラリの使い方
- cursesの考え方
- 基本入出力制御
- 複数のウィンドウ
- キーパッド
- カラー

この章では、これまでに学んだことを応用し、CDデータベースプログラムを完全にCで書き直します。

6.1 cursesを使ったプログラムのコンパイル

cursesはライブラリです。このため、cursesを使うには、適切なヘッダーファイルのインクルードとライブラリの指定が必要になります。

cursesライブラリには、歴史的経緯からいくつかの異なる実装が存在します。cursesはもともとBSD UNIXで登場したものがオリジナルで、これがSystem V系のUNIXにも取り込まれま

した。この章のサンプルプログラムで利用しているncursesは、System V Relase 4.0 cursesのフリーウェアエミュレーションで、Linuxで開発されたものです。ncursesは、ほかのUNIXシステムに対する移植性にも優れているので、Linux以外のUNIXシステムに付属するcursesでサポートされていない機能がある場合には、代わりにncursesを利用するのもひとつの方法です。

メモ

X/Open仕様では、cursesに"基本"と"拡張"の2つのレベルを定義しています。ncursesのバージョン1.9.9では、このうち拡張機能の一部がまだ実装されていません。ただし、ncursesに実装されていない拡張機能が必要になることはあまりありません。
拡張cursesには、マルチカラム文字を処理するためのルーチンやカラー操作ルーチンなど、追加のルーチンが含まれています。

cursesを使ったプログラムをコンパイルする場合、通常はcurses.hをインクルードし、-lcursesを指定してcursesライブラリとリンクします。ただし、システムによってはすでにncursesを使うように設定されている場合があります。筆者のマシンでは、最新バージョンのcursesを使用するために、/usr/include/ncursesディレクトリにあるヘッダーファイルcurses.hをインクルードし、-lncursesを指定してncursesライブラリとリンクするようにしました。コンパイラにはGNU Cコンパイラgccを使っているので、コンパイル時のコマンドラインは次のようになります。

```
$ gcc -I/usr/include/ncurses program.c -o program -lncurses
```

-Iオプションを使って、ヘッダーファイルを検索するディレクトリを指定している点に注意してください。

cursesがシステムにどのようにセットアップされているかわからない場合には、ncurses関連のマニュアルページやディストリビューションのドキュメントを参照してください。また、システム管理者が別にいる場合には、システム管理者に問い合わせてください。

6.2 cursesの考え方

プログラムの記述を始める前に、cursesの基本的な考え方について説明しておきましょう。

cursesの各種ルーチンは、画面、ウィンドウ、サブウィンドウを対象として動作します。画面（スクリーン）とは、書き込みの対象となるデバイス（通常は端末画面）のことです。画面は、そのデバイス上の利用可能な表示部分すべてに対応します。たとえば、X Window上の端末ウィンドウの場合、その端末ウィンドウの内側で文字を配置できるすべての領域が画面になります。cursesでは、stdscrと呼ばれるcursesウィンドウが少なくとも必ず1つ存在します。stdscr

は、物理画面と同じサイズです。画面よりもサイズの小さい追加のウィンドウを作成することもできます。これらのウィンドウは互いにオーバーラップ可能で、複数のサブウィンドウを持つことができます。ただし、各サブウィンドウは、常に親ウィンドウの内側になければなりません。

cursesライブラリは、端末画面のマップとして機能する2つのデータ構造体、stdscrとcurscrを管理します。

このうち、より重要性の高いstdscrは、cursesの関数が出力を生成するときに更新される構造体です。stdscrデータ構造体は**標準画面**（standard screen）を表します。stdscrは、stdioライブラリでの標準出力stdoutとちょうど同じような機能を果たし、cursesを使ったプログラムでのデフォルトの出力ウィンドウになります。

ただし、出力は、プログラムの中でrefreshを呼び出すまで画面には表示されません。refreshが呼び出されると、cursesライブラリはstdscrの内容（表示すべき画面）と、もうひとつの構造体curscr（現在の画面の状態）とを比較します。そして、これら2つの構造体の差分を使って画面を更新します。

cursesを使った一部のプログラムでは、cursesがstdscr構造体を保持していることを意識する必要があります。これは、少数のcurses関数でstdscr構造体がパラメータとして必要になるからです。ただし、stdscr構造体自体は処理系依存であり、決して直接アクセスしてはいけません。cursesを利用するプログラムでは、curscrを使う必要はまずありません。

cursesを使ったプログラムでの文字出力の処理は、次のようになります。

- cursesの関数を使って**論理画面**を更新する。
- refreshを使用し、cursesに**物理画面**の更新を要求する。

cursesのこうした2段階のアプローチには、画面の更新を非常に効率的に行うことができるという利点があります。この利点は、コンソール画面を使うかぎりではあまり実感できないかもしれません。しかし、低速のシリアル回線やモデム経由でプログラムを実行する場合には大きな威力を発揮します。

通常、cursesを使うプログラムでは、論理画面への出力を行う関数を何度か呼び出し、画面のあちこちに移動してテキストを書き込んだり、線やボックスを描画したりします。しかし、これだけでは実際の画面上にはなにも表示されません。ある段階に至ったら、ユーザーにそれまでの出力結果をすべて見せる必要があります。こうした必要性が生じると（一般にはrefreshが呼び出されたとき）、cursesは物理画面を論理画面に対応させるための最適な方法を計算します。cursesでは、適切な端末機能を使用し、カーソル移動を最適化することによって、プログラムで要求されたものよりはるかに少ないカーソル移動量で画面を更新することができます。cursesライブラリの名前自体も、このカーソル最適化機能からとられています。

論理画面のレイアウトは、画面左上隅を画面位置(0,0)とし、行とカラムに分けて配置される文字の配列になっています。

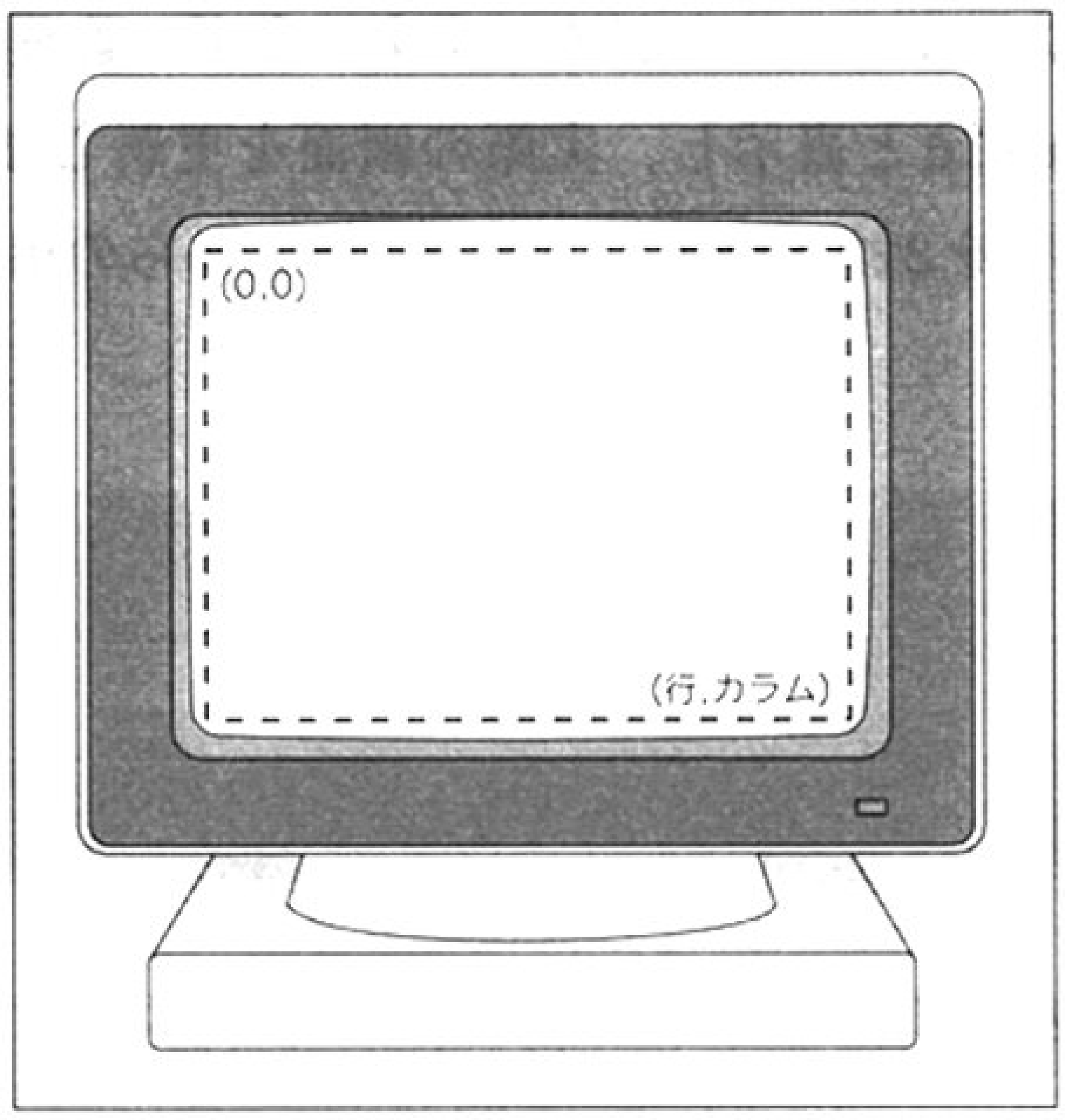


図 6.1 論理画面のレイアウト

cursesライブラリのすべての関数は、行の値（y）を前に、カラムの値（x）をあとに指定する座標系を使います。

各位置は、その位置にある文字だけでなく、文字の属性も保持しています。表示可能な属性は物理端末の機能によって異なりますが、ふつうは少なくとも太字と下線の表示が可能です。

cursesライブラリでは、いくつかの一時的なデータ構造を作成したり破棄したりする必要があるので、cursesを使うプログラムでは、まずライブラリの初期化を行い、使用後にはもとの設定を復元できるようにしなければなりません。この処理は、対となる2つの関数呼び出しinitscrとendwinによって行います。

では、さっそくcursesを使って簡単なプログラム、screen1.cを作成してみましょう。このプログラムで処理の流れと基本的な関数呼び出しの使い方を示したあと、cursesライブラリの各関数について詳しく説明します。

例題 cursesを使った簡単なプログラム

1 ヘッダーファイルcurses.hをインクルードし、main関数でcursesライブラリの初期化とリセットを行う文を記述します。

```
#include <unistd.h>
#include <curses.h>

int main() {
    initscr();

...

    endwin();
    exit(0);
}
```

2 上の初期化とリセットの間には、次のようなコードを記述します。まず、論理画面上の位置(5,15)にカーソルを移動して`Hello World`と表示し、実際の画面を更新します。最後に、`sleep(2)`を呼び出してプログラムの実行を2秒間停止し、プログラムからの出力を確認できるようにします。

```
move(5, 15);
printw("%s", "Hello World")
refresh();

sleep(2);
```

プログラムを実行すると、空白の画面の左上のほうに`Hello World`と表示されます。

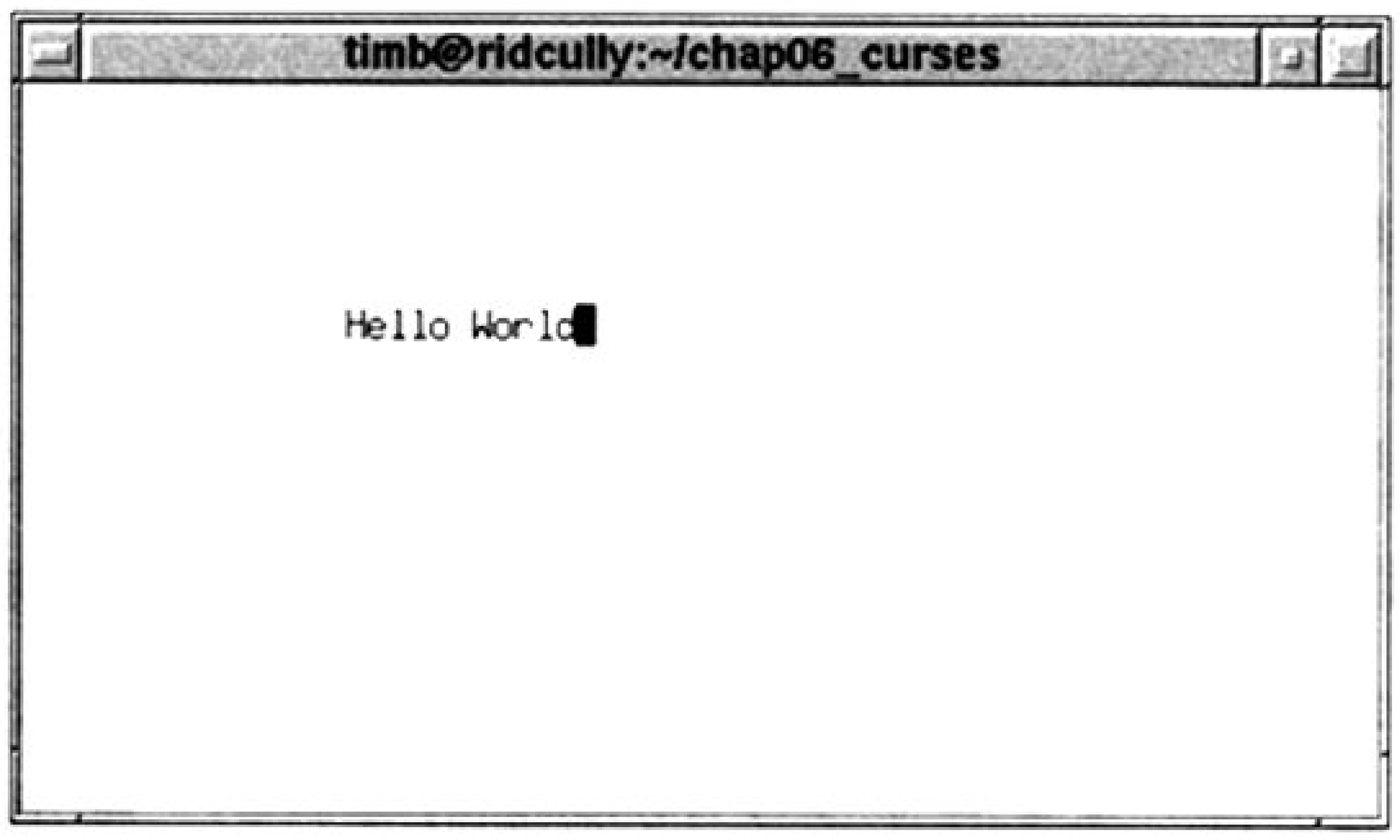


図 6.2 cursesを使った簡単なプログラム

6.3 基本入出力機能

ここでは、画面表示のためのcursesの主な機能について説明します。

6.3.1 初期化と終了

すでに説明したように、cursesを使うプログラムでは、必ず最初に`initscr`を呼び出し、最後に`endwin`を呼び出す必要があります。必要なヘッダーファイルと構文は次のとおりです。

```
#include <curses.h>

WINDOW *initscr(void);
int endwin(void);
```

initscr関数は、プログラムの中で一度だけ呼び出します。**initscr**関数は、成功すると**stdscr**構造体へのポインタを返します。失敗した場合にはエラーメッセージを表示し、プログラムを終了します。

endwin関数は、成功すると**OK**を返し、失敗すると**ERR**を返します。**endwin**を呼び出していったん**curses**を離れ、あとで**clearok(stdscr, 1)**と**refresh**を呼び出すと、**curses**での操作を再開することができます。この場合、**curses**が保持していた物理画面の内容は破棄され、画面は完全に再描画されます。

WINDOW構造体は、**curses**が画面表示を格納するために使う構造体です。**curses**では、この構造体の内部にアクセスすることを許可していません。

6.3.2 出力

次に示すのは、画面を更新するための基本関数です。

```
#include <curses.h>

int addch(const chtype char_to_add);
int addchstr(chtype *const string_to_add);
int printw(char *format, ...);
int refresh(void);
int box(WINDOW *win_ptr, chtype vertical_char, chtype horizontal_char);
int insch(chtype char_to_insert);
int insertln(void);
int delch(void);
int deleteln(void);
int beep(void);
int flash(void);
```

cursesでは、**chtype**という独自の文字型を使います。**chtype**は、通常の**char**型よりビット数が多い場合があります。Linuxで使われているバージョンの**ncurses**の**chtype**は、実際には**unsigned long**です。

名前の最初に**add**が付く関数は、指定された文字または文字列を現在の位置に追加します。**printw**関数は、**printf**と同様の方法で文字列に書式を設定し、結果の文字列を現在の位置に追加します。

refresh関数は物理画面を更新します。成功すると**OK**を返し、失敗すると**ERR**を返します。

box関数を使うと、ウィンドウに沿ってボックスを描くことができます。ただし、標準の**curses**では、垂直と水平の線を描く文字として通常の文字しか使用できないことがあります。

メモ

拡張**curses**では、**ACS_VLINE**と**ACS_HLINE**を使うことで、見ばえのよいボックスを描くことができます。この場合、端末が線描画文字をサポートしている必要がありますが、現在ではほとんどの端末がこれらの文字をサポートしています。

insch関数は1文字を挿入し、それ以降の既存の文字を右へ移動します。ただし、行の終わりの処理については指定がなく、使用している端末によって左右されます。**insertln**は空行を1つ挿入し、既存の行を1つずつ下へ移動します。

2つの**delete**関数は、それぞれ対応する**insert**関数と同様の動作をします。

beepを呼び出すと、音を鳴らすことができます。ごく少数の端末は音を出すことができません。この場合、**curses**の設定によっては、画面をフラッシュさせることになります。周囲に多数のマシンがあり、あちこちからビープ音が聞こえてくるような環境では、音を鳴らすより画面をフラッシュさせるほうが好ましいかもしれません。**flash**は画面をフラッシュさせます。端末にこの機能がない場合、**curses**は代わりに音を鳴らそうとします。

6.3.3 画面の読み取り

次に示す関数を使うと、画面から文字を読み取ることができます。ただし、この機能はあまり使われません。

```
#include <curses.h>

chtype inch(void);
int instr(char *string);
int innstr(char *string, int number_of_characters);
```

inch関数は必ず用意されていますが、**instr**関数と**innstr**関数はサポートされているとは限りません。**inch**関数は、現在カーソルがある画面位置の文字とその属性情報を返します。**inch**が返すのは**char**ではなく、**chtype**です。**instr**と**innstr**は、**char**の配列に書き込むことに注意してください。

6.3.4 クリア

画面の領域をクリアするには、主に次の4つの関数を使います。

```
#include <curses.h>

int erase(void);
int clear(void);
int clrtobot(void);
int clrtoeol(void);
```

erase関数は、画面内のすべての位置に空白を書き込みます。**clear**関数は、**erase**関数同様に画面をクリアしますが、**clearok**も呼び出して画面が再描画されるようにします。**clearok**

は、次にrefreshが呼び出されたときに画面のクリアと再描画が行われるようにします。

通常、clear関数は、画面上の現在空白ではない位置を消去するのではなく、画面全体を消去する端末コマンドを使用します。したがって、clear関数は、画面を完全に消去するための確実な方法として使用することができます。clearのあとにrefreshを実行する方法は、再描画コマンドとして便利です。

clrtobotはカーソル位置以降の画面をクリアし、clrtoeolはその行のカーソルから行の終わりまでをクリアします。

6.3.5 移動

カーソルを移動する関数は1つだけですが、ほかに画面更新後のカーソルの位置を制御するための関数が用意されています。

```
#include <curses.h>

int move(int new_y, int new_x);
int leaveok(WINDOW *window_ptr, bool leave_flag);
```

move関数は、論理カーソルの位置を指定された位置に移動します。画面の座標系は、左上隅を(0,0)として指定します。ほとんどのバージョンのcursesでは、2つのextern整数LINESとCOLUMNSに物理画面のサイズが収められており、これらの変数を使うことで、new_yとnew_xに指定できる最大値を調べることができます。

moveを呼び出しても、それだけでは物理カーソルは移動されないことに注意する必要があります。moveは、論理画面上で次の出力を表示する位置を変更するだけです。moveを呼び出したすぐ後に画面カーソルも移動したい場合には、refreshを呼び出します。

leavok関数は、画面更新後の物理カーソルをどこに置くかを制御するフラグを設定します。このフラグはデフォルトではfalseで、更新後のハードウェアカーソルは画面上で論理画面と同じ位置に置かれます。フラグがtrueに設定されている場合、ハードウェアカーソルは画面上のいずれかの場所にランダムに置かれます。

6.3.6 属性

cursesでは、各文字がいくつかの属性を持つことができます。これらの属性は、該当する属性を表示ハードウェアがサポートしている場合に、画面上への文字の表示方法を表すものです。定義済みの属性には次のものがあります。

A_BLINK、A_BOLD、A_DIM、A_REVERSE、A_STANDOUT、A_UNDERLINE

属性を単独で、またはまとめて設定するには、次の関数を使います。

```
#include <curses.h>

int attron(chtype attribute);
int attroff(chtype attribute);
int attrset(chtype attribute);
int standout(void);
int standend(void);
```

attrset関数は、**curses**の属性を設定し、**attron**と**attroff**は、指定された属性をほかの属性には影響を与えずに有効または無効にします。**standout**と**standend**は、汎用的な強調モードである"スタンドアウトモード"を利用するための関数です。ほとんどの端末では、スタンドアウトモードは反転表示にマップされます。

以上で画面の管理方法についてひととおり学んだので、実際にサンプルプログラムを作成してみましょう。作成するプログラムの名前は**moveadd.c**とします。プログラムでは、画面がどのように表示されるかを確認できるようにするために、**refresh**と**sleep**を何度か呼び出します。通常、**curses**を使うプログラムでは、画面の更新をできるだけ行わないようにします。これは、画面の更新があまり効率のよい操作ではないからです(サンプルプログラムでは、効率的に不利になることを承知のうえで、あえて画面の更新を行っています)。

例題 移動、挿入、文字の属性

1 まず、必要なヘッダーファイルをインクルードし、いくつかの文字配列とそれらへのポインタを定義したあと、**curses**を初期化します。

```
#include <unistd.h>
#include <curses.h>

int main()
{
    const char witch_one[] = " First Witch  ";
    const char witch_two[] = " Second Witch ";
    const char *scan_ptr;

    initscr();
```

2 テキストの属性をオンやオフにしながら、間隔をおいて画面上に3つのテキストを表示します。

```
    move(5, 15);
    attron(A_BOLD);
    printw("%s", "Macbeth");
    attroff(A_BOLD);
    refresh();
```

```
    sleep(1);

    move(8, 15);
    attron(A_DIM);
    printw("%s", "Thunder and Lightning");
    attroff(A_DIM);
    refresh();
    sleep(1);

    move(10, 10);
    printw("%s", "When shall we three meet again");
    move(11, 23);
    printw("%s", "In thunder, lightning, or in rain ?");
    move(13, 10);
    printw("%s", "When the hurlyburly's done,");
    move(14,23);
    printw("%s", "When the battle's lost and won.");
    refresh();
    sleep(1);
```


3 最後に、セリフをしゃべっている登場人物の名前を1文字ずつ挿入します。mainの関数の最後では、endwin関数を呼び出します。

```
    attron(A_DIM);
    scan_ptr = witch_one + strlen(witch_one);
    while(scan_ptr != witch_one) {
        move(10,10);
        insch(*scan_ptr--);
    }

    scan_ptr = witch_two + strlen(witch_two);
    while (scan_ptr != witch_two) {
        move(13, 10);
        insch(*scan_ptr--);
    }
    attroff(A_DIM);

    refresh();
    sleep(1);

    endwin();
    exit(0);
}
```

次の図は、プログラムの最後に表示される画面です。

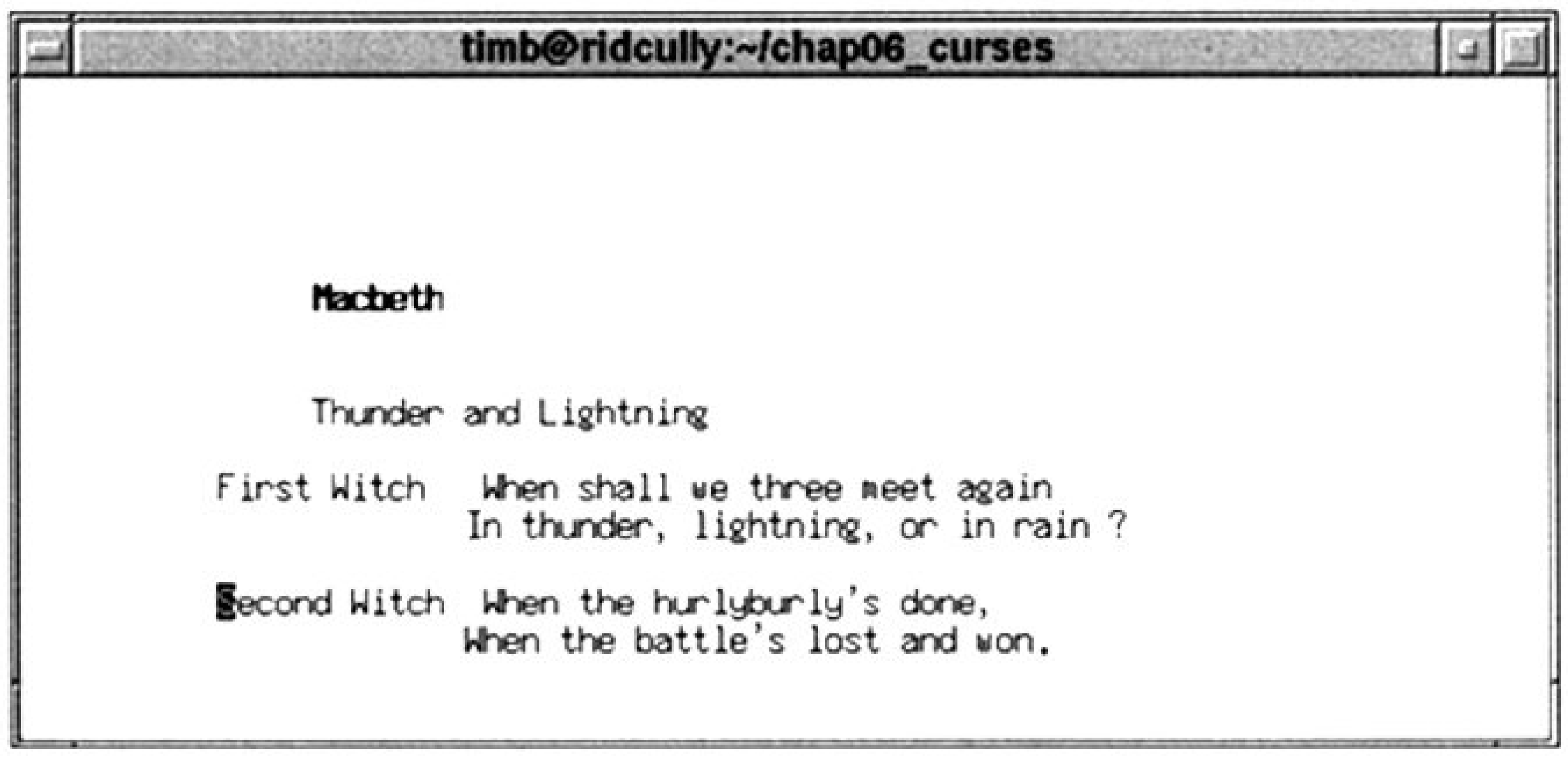


図 6.3 移動、挿入、文字の属性

6.4 キーボード

cursesでは、画面制御用のインタフェースだけでなく、キーボードを制御するための方法も用意されています。

6.4.1 キーボードのモード

キーボード読み取りルーチンは、モードによって制御されます。モードを設定する関数は、次のとおりです。

```
#include <curses.h>

int echo(void);
int noecho(void);
int cbreak(void);
int nocbreak(void);
int raw(void);
int noraw(void);
```

最初の2つの**echo**関数は、入力文字のエコーを有効または無効にします。残りの4つの関数は、端末に入力された文字を**curses**プログラムでどのように利用できるようにするかを指定します。

cbreakについて説明する前に、デフォルトの入力モードについて理解しておく必要があります。**initscr**を呼び出し、**curses**を使うプログラムが起動すると、入力モードはいわゆる**Cooked**モードに設定されます。Cookedモードでは、すべての処理が行単位で行われ、ユーザーが**Return**キーを押した時点で初めて入力が利用可能になります。キーボードの特殊文字は有

効になっているので、しかるべきキーの組み合わせを押せば、プログラムでシグナルを発生することができます。フロー制御も有効になっています。プログラムでcbreakを呼び出すと、入力モードをCbreakモードに設定することができます。Cbreakモードでは、入力した文字が即座にプログラムから利用可能になります。Cookedモード同様、Cbreakモードでもキーボードの特殊文字は有効になっています。

rawを呼び出すと、特殊文字の処理が無効になり、特殊文字の組み合わせを入力しても、シグナルを発生したりフロー制御を行ったりすることはできなくなります。nocbreakを呼び出すと、入力モードはCookedモードに戻されますが、特殊文字の処理については変更されません。norawを呼び出すと、Cookedモードと特殊文字の処理の両方がもとの設定に戻されます。

6.4.2 キーボード入力

キーボードの読み取りは非常に簡単です。主に使う関数は次のとおりです。

```
#include <curses.h>

int getch(void);
int getstr(char *string);
int getnstr(char *string, int number_of_characters);
int scanw(char *format, ...);
```

これらの関数は、標準入出力ライブラリのgetchar、gets、scanfと非常によく似た機能を持っています。getstrは、返される文字列の長さを制限することができないので、十分慎重に使う必要があります。使用しているcursesのバージョンが、読み取る文字数を制限できるgetnstrをサポートしている場合には、getstrではなくgetnstrを使うほうがよいでしょう。両者の動作の違いは、第3章で説明したgetsとfgetsの違いとよく似ています。

例題 キーボードのモードと入力

1 まず必要な初期設定を行って、cursesの関数をいくつか呼び出します。

```
#include <unistd.h>
#include <curses.h>
#include <string.h>

#define PW_LEN 25
#define NAME_LEN 256

int main() {
    char name[NAME_LEN];
    char password[PW_LEN];
    char *real_password = "xyzzy";
```

```
    int i = 0;

    initscr();

    move(5, 10);
    printw("%s", "Please login:");

    move(7, 10);
    printw("%s", "User name: ");
    getstr(name);

    move(9, 10);
    printw("%s", "Password: ");
    refresh();
```

2 ユーザーがパスワードを入力するときに、パスワードが画面にエコーされないようにする必要があります。受け取ったパスワードは、xyzzyと比較します。

```
    cbreak();
    noecho();

    memset(password, '\0', sizeof(password));
    while (i < PW_LEN) {
        password[i] = getch();
        move(9, 20 + i);
        addch('*');
        refresh();
        if (password[i] == '\n') break;
        if (strcmp(password, real_password) == 0) break;
        i++;
    }
```

3 最後にキーボードエコーを有効にして、入力されたパスワードが正しいかどうか表示します。

```
    echo();
    nocbreak();

    move(11, 10);
    if (strcmp(password, real_password) == 0) printw("%s", "Correct");
    else printw("%s", "Wrong");
    refresh();

    endwin();
    exit(0);
}
```

解説

キーボード入力のエコーを無効にして、入力モードをCbreakモードに設定し、パスワードを受け取るためのメモリ領域を用意しています。パスワードの各文字は入力と同時に処理され、画面にはアスタリスク（*）が表示されます。画面は、文字が入力されるたびに更新する必要があります。パスワードを受け取ったら、strcmpを使って、あらかじめプログラムの中で設定しておいたパスワードと比較します。

注意

> 古いバージョンのcursesライブラリを使っている場合には、getstrを呼び出す前に明示的にrefreshを実行するなど、プログラムを一部変更する必要があるかもしれません。新しいcursesでは、getstrはgetchを呼び出すように定義されており、画面の更新もgetchで行われます。

6.5 複数のウィンドウ

これまでは、端末をフル画面出力の媒体として扱ってきました。小さいシンプルなプログラムの場合にはこれでも十分ですが、cursesライブラリでは、物理画面上にサイズの異なる複数のウィンドウを同時に表示するといった複雑な処理も可能です。

これから取り上げる関数の多くは、X/Openの用語による“拡張curses”でのみサポートされています。しかし、これらの関数はncursesでもサポートされており、ほとんどのプラットフォームで問題なく使用できます。

以下では複数のウィンドウの使い方について説明します。また、これまでに学んだコマンドが、複数のウィンドウを使う場合を考慮してどのように汎用化されているかについても学びます。

6.5.1 WINDOW構造体

標準画面stdscrについてはすでに少し触れましたが、これまではほとんど使う機会がありませんでした。今までに使用した関数はstdscrを対象として動作するものなので、stdscrをパラメータとして渡す必要がなかったためです。

stdscrは、WINDOW構造体のひとつの特殊なケースにすぎません。これは、stdoutがファイルストリームの特別なケースであるのと同じです。通常、WINDOW構造体はcurses.hで宣言されています。WINDOW構造体を調べるといろいろなことがわかって興味深いのですが、WINDOW構造体は処理系によって異なるので、プログラムから直接WINDOW構造体にアクセスしてはいけません。

ウィンドウを作成または削除するには、**newwin**関数と**delwin**関数を使います。

```
#include <curses.h>

WINDOW *newwin(int num_of_lines, int num_of_cols, int start_y, int start_x);
int delwin(WINDOW *window_to_delete);
```

newwin関数は、指定された行数とカラム数の新しいウィンドウを画面位置(**start_y, start_x**)に作成します。作成に成功すると、新しいウィンドウへのポインタを返し、失敗した場合には**NULL**を返します。新しいウィンドウの右下隅を画面の右下隅に合わせたい場合には、行数またはカラム数に0を指定します。

ウィンドウは、いずれも現在の画面の中に収まっていなければなりません。したがって、新しいウィンドウに画面領域からはみだす部分があると、**newwin**は失敗します。

delwin関数は、**newwin**によって作成されたウィンドウを削除します。**newwin**が呼び出されると、新しいウィンドウ用のメモリが割り当てられることになるので、不要になったウィンドウは必ず削除しておきます。ただし、**curses**自身のウィンドウである**stdscr**と**curscr**は削除しないようにしてください。

newwinによって作成された新しいウィンドウは、既存のすべてのウィンドウから完全に独立しています。デフォルトでは、新しいウィンドウは既存のどのウィンドウよりも手前に表示され、ほかのウィンドウの内容は新しいウィンドウによって隠されることになります（内容は変更されません）。

さて、新しいウィンドウを作成したあと、どのような方法でウィンドウに書き込みを行ったらよいのでしょうか。実は、これまでに学んだ大部分の関数には、指定されたウィンドウを対象として動作する汎用バージョンが用意されています。これらの関数は、使いやすいようにカーソル移動も含めて指定できるようになっています。

6.5.2 汎用バージョンの関数

これまでのサンプルプログラムでは、**addch**と**printw**を使って画面に文字を追加しました。これらの関数を含めて、ほとんどの関数では、プレフィックス**w**、**mv**、または**mvw**を関数名の頭に付けることができます。ここで、**w**は“ウィンドウ”（window）を表し、**mv**は“移動”（move）、**mvw**は“移動”と“ウィンドウ”（move and window）を表します。

ほとんどの**curses**の実装では、これまでに使用した関数の多くがヘッダーファイル**curses.h**でマクロ（**#define**）によって定義されており、実際には汎用バージョンの関数を呼び出すようになっています。

プレフィックス**w**が付いている場合には、**WINDOW**構造体へのポインタを引数リストに含める必要があります。プレフィックス**mv**が付いている場合には、2つの追加パラメータ**y**と**x**を含め

る必要があります。これらのパラメータは、操作を実行する位置を指定するもので、画面ではなくウィンドウの左上隅を(0,0)とする相対位置で指定します。

プレフィックス**mvw**が付いている場合には、**WINDOW**構造体へのポインタ、および**y**と**x**の3つの追加パラメータを引数リストに含める必要があります。**mvw**というプレフィックスからは、引数の順序として**y**と**x**が先に来るように思えるかもしれませんが、実際には座標指定の前に**WINDOW**構造体へのポインタを指定します。

例として、**addch**と**printw**の汎用バージョンのプロトタイプ宣言を次に示します。

```
#include <curses.h>

int addch(const chtype char);
int waddch(WINDOW *window_pointer, const chtype char)
int mvaddch(int y, int x, const chtype char);
int mvwaddch(WINDOW *window_pointer, int y, int x, const chtype char);
int printw(char *format, ...);
int wprintw(WINDOW *window_pointer, char *format, ...);
int mvprintw(int y, int x, char *format, ...);
int mvwprintw(WINDOW *window_pointer, int y, int x, char *format, ...);
```

その他の多くの関数、たとえば**inch**などでも、移動やウィンドウを指定できるバージョンが用意されています。

6.5.3 ウィンドウの移動と更新

実際に複数のウィンドウを使う前に、ウィンドウを操作するための関数についても見ておきましょう。これらの関数は、ウィンドウを移動したり再描画したりするときに使います。

```
#include <curses.h>

int mvwin(WINDOW *window_to_move, int new_y, int new_x);
int wrefresh(WINDOW *window_ptr);
int wclear(WINDOW *window_ptr);
int werase(WINDOW *window_ptr);
int touchwin(WINDOW *window_ptr);
int scrollok(WINDOW *window_ptr, bool scroll_flag);
int scroll(WINDOW *window_ptr);
```

mvwin関数は、画面上のウィンドウを移動します。1つのウィンドウの各部分は画面領域の中に収まっていなければならないので、ウィンドウのある部分が画面領域の外にはみだすような形でウィンドウを移動すると、**mvwin**は失敗します。

wrefresh、**wclear**、**werase**は、それぞれすでに説明した**refresh**、**clear**、**erase**の汎用バージョンで、**stdscr**ではなく特定のウィンドウを対象として操作が行えるように、**WINDOW**構造体へのポインタを引数に取ります。

touchwinは特別な関数です。この関数は、パラメータによって指定されたウィンドウの内容が変更されていることをcursesに知らせる機能を持っています。つまり、touchwinを呼び出すと、次にwrefreshが呼び出されたときに、ウィンドウの内容が実際には変更されていなくても必ずウィンドウが再描画されます。touchwinは、画面上でオーバーラップして表示されている複数のウィンドウの中から、手前に表示するウィンドウを指定する場合に便利な関数です。

2つのスクロール関数は、ウィンドウのスクロールを制御します。scrollok関数はブール型の引数を取り、TRUE（通常は0以外）を指定すると、ウィンドウはスクロール可能になります。デフォルトではウィンドウはスクロールできません。scroll関数は、ウィンドウを1行上にスクロールさせます。cursesの実装によっては、スクロールの行数を（負の値を含めて）指定できるwscrl関数が用意されていることがあります。スクロールについては、あとでもう一度取り上げます。

さて、複数のウィンドウの扱い方についても学びましたから、新しい関数を使ってサンプルプログラムを作成してみましょう。名前はmultiw1.cとします。なお、コードが複雑になるのを避けるため、エラーチェックは省いています。

例題　複数のウィンドウ

1 まずヘッダーファイルをインクルードし、必要な定義を記述します。

```
#include <unistd.h>
#include <curses.h>

int main()
{
    WINDOW *new_window_ptr;
    WINDOW *popup_window_ptr;
    int x_loop;
    int y_loop;
    char a_letter = 'a';

    initscr();
```

2 ベースウィンドウに文字を書き込みます。論理画面に文字を書き終えたら、実際の画面を更新します。

```
    move(5, 5);
    printw("%s", "Testing multiple windows");
    refresh();

    for (x_loop = 0; x_loop < COLS - 1; x_loop++) {
        for (y_loop = 0; y_loop < LINES - 1; y_loop++) {
            mvwaddch(stdscr, y_loop, x_loop, a_letter);
            a_letter++;
            if (a_letter > 'z') a_letter = 'a';
```

```
        }
    }

    refresh();
    sleep(2);
```

3 次に、新しく10x20のウィンドウを作成し、このウィンドウにテキストを追加してから画面に描画します。

```
    new_window_ptr = newwin(10, 20, 5, 5);
    mvwprintw(new_window_ptr, 2, 2, "%s", "Hello World");
    mvwprintw(new_window_ptr, 5, 2, "%s",
              "Notice how very long lines wrap inside the window");

    wrefresh(new_window_ptr);
    sleep(2);
```


4 今度はバックグラウンドウィンドウの内容を変更します。画面を更新すると、new_window_ptrで示されるウィンドウがバックグラウンドウィンドウによって隠されます。

```
    a_letter = '0';
    for (x_loop = 0; x_loop < COLS - 1; x_loop++) {
        for (y_loop = 0; y_loop < LINES -1; y_loop++) {
            mvwaddch(stdscr, y_loop, x_loop, a_letter);
            a_letter++;
            if (a_letter > '9') a_letter = '0';
        }
    }


    refresh();
    sleep(2);
```

5 新しく作成したウィンドウを更新します。ウィンドウの内容は変更されていないので、画面は変化しません。

```
    wrefresh(new_window_ptr);
    sleep(2);
```

6 しかし、いったんウィンドウをタッチし、ウィンドウの内容が変更されたことをcursesに通知してやると、次にwrefreshを呼び出したときに、新しいウィンドウが再度手前に表示されます。

```
    touchwin(new_window_ptr);
    wrefresh(new_window_ptr);
    sleep(2);
```

7 次に、オーバーラップするウィンドウを1つ追加します。今度はウィンドウに沿ってボックスを描きます。

```
    popup_window_ptr = newwin(10, 20, 8, 8);
    box(popup_window_ptr, '|', '-');
    mvwprintw(popup_window_ptr, 5, 2, "%s", "Pop Up Window!");
    wrefresh(popup_window_ptr);
    sleep(2);
```

8 最初に作成した新しいウィンドウと、上の手順で作成したポップアップウィンドウとを使ってさまざまな操作を実行します。

```
    touchwin(new_window_ptr);
    wrefresh(new_window_ptr);
    sleep(2);

    wclear(new_window_ptr);
    wrefresh(new_window_ptr);
    sleep(2);

    delwin(new_window_ptr);

    touchwin(popup_window_ptr);
    wrefresh(popup_window_ptr);
    sleep(2);

    delwin(popup_window_ptr);

    touchwin(stdscr);
    refresh();
    sleep(2);

    endwin();
    exit(0);
}
```

プログラムを実行したときのようすは、残念ながら紙面ではうまく伝えることができませんが、途中で表示される画面のひとつを次に示しておきます。

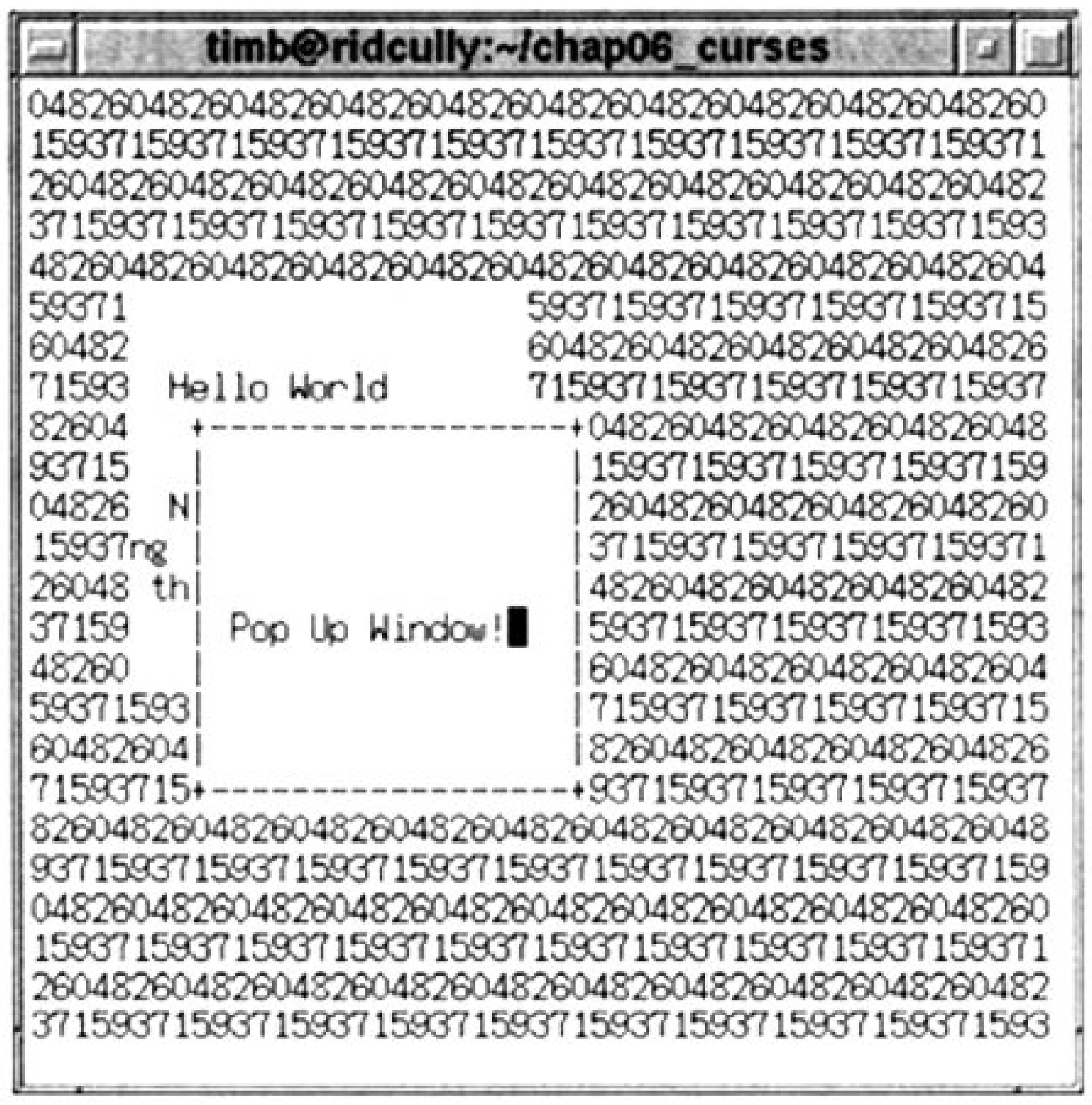


図 6.4　複数のウィンドウ

解説

サンプルプログラムからわかるように、複数のウィンドウを更新する場合には、期待した順序でウィンドウを画面に表示するための工夫が必要です。**curses**に対してウィンドウの更新を要求した場合、ウィンドウの階層構造に関する情報は**curses**の処理の対象になりません。つまり、ウィンドウが**curses**によって正しい順序で描画されるようにするには、正しい順序でウィンドウを更新する必要があります。そのためのひとつの方法として、ウィンドウへのポインタをすべて配列またはリストに格納しておき、この配列またはリストを目的の順序に従って並べ替えるやり方があります。

6.5.4　画面更新の最適化

上のサンプルプログラムで示したように、複数のウィンドウを更新するには注意が必要ですが、それほど複雑な操作が要求されるわけではありません。ただし、モデムなどの低速な回線に接続された端末を更新する場合には、画面の更新処理が問題となることがあります。

低速な回線では、画面への描画が遅くなり、操作感を大きく損なう可能性があるので、描画する文字数をできるだけ抑える必要があります。**curses**では、こうしたケースに対処するための特別な方法として、**wnoutrefresh**と**doupdate**が用意されています。

```
#include <curses.h>

int wnoutrefresh(WINDOW *window_ptr);
int doupdate(void);
```

wnoutrefresh関数は、画面に送信する必要がある文字がどれかを決定しますが、実際に文字

を画面に送信することはしません。**doupdate**関数は、変更内容を実際に端末に送信します。

wnoutrefreshのすぐあとで**doupdate**を呼び出した場合、結果的には**wrefresh**を呼び出したのと同じになります。しかし、複数のウィンドウを再描画する場合には、各ウィンドウに対して**wnoutrefresh**を（正しい順序で）呼び出し、最後に**doupdate**を呼び出すと、各ウィンドウの画面更新処理が**curses**によって計算されたあと、最終的な更新後の画面だけが出力されるので、画面に送信する文字を最小限に抑えることができます。

6.6 サブウィンドウ

サブウィンドウは、複数のウィンドウの特殊なケースです。サブウィンドウを作成または削除するには、次の関数を使います。

```
#include <curses.h>

WINDOW *subwin(WINDOW *parent, int num_of_lines, int num_of_cols,
               int start_y, int start_x);
int delwin(WINDOW *window_to_delete);
```

subwin関数のパラメータリストは、**newwin**関数のものとほぼ同じです。サブウィンドウを削除するには、ほかのウィンドウの場合と同様、**delwin**を使います。サブウィンドウに対する書き込みには、新しいウィンドウの場合と同じように一連の**mvw**関数を使用できます。実際、サブウィンドウの動作は、ほとんどの点で新しいウィンドウと同じです。ただし、ひとつだけ大きく異なる点があります。

その違いとは、サブウィンドウ自身は独自に画面の文字を格納することはしないという点です。サブウィンドウは、サブウィンドウを作成するときに指定した親ウィンドウと文字格納領域を共有します。したがって、サブウィンドウに対して行った変更は親ウィンドウにも反映され、サブウィンドウを削除しても画面は変化しません。

一見したところ、このような動作をするサブウィンドウは不要ではないかと思えます。画面を変更したい場合には、親ウィンドウを変更すればすむからです。しかし、サブウィンドウは、ほかのウィンドウの一部をスクロールしたい場合にその真価を発揮します。

cursesを使ったプログラムでは、しばしば画面の一部をスクロールさせる必要があります。このような場合、目的の部分をサブウィンドウにして、サブウィンドウをスクロールさせると、期待した結果を得ることができます。

注意

> サブウィンドウを使う場合、ひとつだけ制限があります。それは、アプリケーション側で親ウィンドウに対してtouchwinを呼び出してから画面を更新する必要があることです。

次に示すのは、前のサンプルプログラムに修正を加えたsubscl.cです。このプログラムでは、親ウィンドウとは独立してスクロールするサブウィンドウを使っています。

例題 サブウィンドウ

1 ベースウィンドウを初期化し、テキストを書き込みます。

```
#include <unistd.h>
#include <curses.h>

int main()
{
    WINDOW *sub_window_ptr;
    int x_loop;
    int y_loop;
    int counter;
    char a_letter = '1';

    initscr();

    for (x_loop = 0; x_loop < COLS - 1; x_loop++) {
        for (y_loop = 0; y_loop < LINES - 1; y_loop++) {
            mvwaddch(stdscr, y_loop, x_loop, a_letter);
            a_letter++;
            if (a_letter > '9') a_letter = '1';
        }
    }
```

2 スクロール用の新しいサブウィンドウを作成し、サブウィンドウを使うときの注意に従って、親ウィンドウをタッチしてから画面を更新します。

```
    sub_window_ptr = subwin(stdscr, 10, 20, 10, 10);
    scrollok(sub_window_ptr, 1);

    touchwin(stdscr);
    refresh();
    sleep(1);
```

3 サブウィンドウの内容を消去し、テキストを書き込んで更新します。次に、ループを使ってテキストをスクロールさせます。

```
    werase(sub_window_ptr);
    mvwprintw(sub_window_ptr, 2, 0, "%s", "This window will now scroll");
    wrefresh(sub_window_ptr);
    sleep(1);

    for (counter = 1; counter < 10; counter++) {
        wprintw(sub_window_ptr, "%s", "This text is both wrapping and \
                    scrolling.");
        wrefresh(sub_window_ptr);
        sleep(1);
    }
```

4 ループを終えたら、サブウィンドウを削除し、ベース画面を更新します。

```
    delwin(sub_window_ptr);

    touchwin(stdscr);
    refresh();
    sleep(1);

    endwin();
    exit(0);
}
```

プログラムの最後のほうで、次のような画面が表示されます。



図 6.5 サブウィンドウ

解説

subwinの戻り値をsub_window_ptrに格納し、サブウィンドウをスクロール可能に設定しています。

サブウィンドウを削除してベースウィンドウ（stdscr）を更新しても、画面上のテキストが変

化しない点に注意してください。これは、**stdscr**の文字データがサブウィンドウによって更新されているためです。

6.7 キーパッド

キーボードを扱うための機能については、すでにいくつか紹介しました。多くのキーボードは、最低でもカーソルキーやファンクションキーを備えています。また、キーパッド（テンキー）や、**Ins**、**Home**といったその他のキーを持つキーボードも少なくありません。

これらのキーのデコードは、多くの端末で難しい問題を投げかけます。というのも、これらのキーは一般にエスケープ文字で始まる文字列を送信するからです。アプリケーションでは、**Esc**キーが1回だけ押された場合と、ファンクションキーによって生成された文字列とを識別しなければならないだけでなく、同じ論理キーに対して端末によって異なるシーケンスが使われる点にも配慮しなければなりません。

cursesでは、ファンクションキーを管理するためのスマートな方法が用意されています。各ファンクションキーによって送信される文字列は、端末ごとに**terminfo**構造体に格納されており、プレフィックス**KEY_**を付けて論理キーを定義した文がインクルードファイル**curses.h**に記述されています。

シーケンスと論理キーとの間の変換は、**curses**が起動した時点では無効になっていますが、**keypad**関数を使って有効にすることができます。**keypad**関数は、成功すると**OK**を返し、失敗すると**ERR**を返します。

```
#include <curses.h>

int keypad(WINDOW *window_ptr, bool keypad_on);
```

パラメータ**keypad_on**に**TRUE**を設定して**keypad**を呼び出し、**Keypad**モードを有効にすると、**curses**でキーシーケンスの処理が行われるようになります。その結果、キーボードの読み取りでは押されたキーが返されるだけでなく、論理キーに対応する**KEY_**定義も返されるようになります。

Keypadモードを使う場合には、3つの問題に注意する必要があります。

まず最初の問題は、エスケープシーケンスの認識がタイミングに左右される点です。多くのネットワークプロトコルは、複数の文字をパケットとしてグループ化したり、逆にひとかたまりの文字を分割したりします。このため、エスケープシーケンスではないものが、グループ化されることによってエスケープシーケンスと認識されてしまったり、エスケープシーケンスが分割され、**Esc**とその他の文字列として認識されてしまうなどの問題が発生する可能性があります。

特にWANや混雑している回線を経由している場合、この問題は深刻です。回避策として唯一

考えられるのは、使用する各ファンクションキーが、重複しない一意の文字を送信するように端末をプログラミングすることです。ただし、この方法では制御文字の数が制限されることになります。

2番目の問題は、Escキーが1回だけ押された場合と、Escで始まるキーボードシーケンスとを区別するために、cursesが短時間待機しなければならない点です。実際、Keypadモードを有効にすると、Escキーの処理にかかるわずかな遅延を体感できることがあります。

最後の問題は、一意ではないエスケープシーケンスをcursesが処理できない点です。同じエスケープシーケンスを送信する2つの異なるキーが端末にある場合、cursesはどちらの論理キーを返せばよいのかわからないので、そのシーケンスを処理しません。

次のkeypad.cは、Keypadモードの使い方を示した短いプログラムです。プログラムを実行してEscキーを押すと、エスケープシーケンスの始まりか、単独で押されたEscキーかを判断するために、わずかながら遅延があることがわかります。

例題 キーパッド

1 プログラムとcursesライブラリの初期化を行い、KeypadモードをTRUEに設定します。

```
#include <curses.h>

#define LOCAL_ESCAPE_KEY    27

int main()
{
    int key;

    initscr();
    crmode();
    keypad(stdscr, TRUE);
```

2 次に、エコーをオフにして、カーソルキーが押されてもカーソルが移動しないようにします。画面をクリアし、テキストを表示します。このあと、プログラムは各キーストロークを受け取り、エラーが発生するか、またはQが押されるまで、入力されたキーを表示します。キーストロークが端末のキーパッドシーケンスと一致する場合には、該当するキーを表示します。

```
    noecho();

    clear();
    mvprintw(5, 5, "Key pad demonstration. Press 'q' to quit");
    move(7, 5);
    refresh();

    key = getch();
    while(key != ERR && key != 'q') {
        move(7,5);
```

```
            clrtoeol();

            if ((key >= 'A' && key <= 'Z') ||
                (key >= 'a' && key <= 'z')) {
                printw("Key was %c", (char)key);
            }
            else {
                switch(key) {
                case LOCAL_ESCAPE_KEY: printw("%s", "Escape key"); break;
                case KEY_END: printw("%s", "END key"); break;
                case KEY_BEG: printw("%s", "BEGINNING key"); break;
                case KEY_RIGHT: printw("%s", "RIGHT key"); break;
                case KEY_LEFT: printw("%s", "LEFT key"); break;
                case KEY_UP: printw("%s", "UP key"); break;
                case KEY_DOWN: printw("%s", "DOWN key"); break;
                default: printw("Unmatched - %d", key); break;
                } /* switch */
            } /* else */

            refresh();
            key = getch();
        } /* while */

        endwin();
        exit(0);
    }
```


6.8 カラー

ごく最近に至るまで、カラー表示をサポートしたダム端末はほとんどなかったので、初期バージョンのcursesはカラーをサポートしていませんでした。しかし、ワークステーションやX端末の普及に伴ってカラー表示機能は標準になり、ncursesや、その他の新しいcursesの実装ではカラーがサポートされるようになっています。

画面上の各文字セルには、一定数のカラーのうちいずれかのカラーを使って書き込みを行うことができます。このとき、バックグラウンドのカラーにも一定数のカラーのうち、いずれかのカラーを使うことができます。たとえば、赤のバックグラウンドに緑でテキストを書き込むことができます。

cursesのカラーサポートは少し変わっていて、文字のカラーを背景と独立した形で定義することはできません。代わりに、文字のフォアグラウンドカラーとバックグラウンドカラーを1つの対として定義しなければなりません。この対のことを、**カラーペア**と呼びます。

cursesでは、カラー機能を使う前に端末がカラーをサポートしているかどうかを調べ、カラ

ールーチンを初期化しなければなりません。そのために用意されているのが次の**has_colors**と**start_color**です。

```
#include <curses.h>

bool has_colors(void);
int start_color(void);
```

has_colorsは、カラーがサポートされていれば**TRUE**を返します。**has_colors**が**TRUE**を返したら、次に**start_color**を呼び出します。**start_color**は、カラーが正常に初期化されると**OK**を返します。

start_colorを呼び出してカラーを初期化すると、変数**COLOR_PAIRS**に、端末がサポートするカラーペアの最大数が設定されます。通常、この値は64です。変数**COLORS**には、利用可能なカラーの最大数が設定されます。通常、この値は8です。

カラーを属性として使う前に、使用するカラーペアを初期化しなければなりません。これには**init_pair**関数を使います。カラー属性は、**COLOR_PAIR**関数を使ってアクセスします。

```
#include <curses.h>

int init_pair(short pair_number, int foreground, int background);
int COLOR_PAIR(int pair_number);
int pair_content(short pair_number, short *foreground, short *background);
```

通常、**curses.h**では、プレフィックス**COLOR_**でいくつかの基本カラーが定義されています。**pair_content**関数を使うと、すでに定義されたカラーペアの情報を得ることができます。

赤をフォアグラウンドカラー、緑をバックグラウンドカラーとしてカラーペア番号1を定義するには、次のようにします。

```
init_pair(1, COLOR_RED, COLOR_GREEN);
```

定義したカラーペアを属性としてアクセスするには、次のように**COLOR_PAIR**を使います。

```
wattron(window_ptr, COLOR_PAIR(1));
```

これで、次に画面に書き込むテキストのフォアグラウンドカラーが赤に、バックグラウンドカラーが緑になります。

COLOR_PAIRは属性のひとつなので、ほかの属性と組み合わせることができます。通常、PCでは、**COLOR_PAIR**と**A_BOLD**のビット単位の論理和を属性に指定することで高輝度カラーを使用できます。

```
wattron(window_ptr, COLOR_PAIR(1) | A_BOLD);
```

さっそくサンプルプログラムを作成してみましょう。名前は**color.c**とします。

例題 カラー

1 まず、プログラムで使用する端末がカラーをサポートしているかどうかを調べます。カラーがサポートされていれば、カラー表示を開始します。

```
#include <unistd.h>
#include <stdio.h>
#include <curses.h>

int main()
{
    int i;

    initscr();
    if (!has_colors()) {
        endwin();
        fprintf(stderr, "Error - no color support on this terminal\n");
        exit(1);
    }

    if (start_color() != OK) {
        endwin();
        fprintf(stderr, "Error - could not initialize colors\n");
        exit(2);
    }
```

2 使用できるカラーとカラーペアの数を表示し、カラーペアを7つ作成して1つずつ画面に表示します。

```
    clear();
    mvprintw(5, 5, "There are %d COLORS, and %d COLOR_PAIRS available",
            COLORS, COLOR_PAIRS);
    refresh();

    init_pair(1, COLOR_RED, COLOR_BLACK);
    init_pair(2, COLOR_RED, COLOR_GREEN);
    init_pair(3, COLOR_GREEN, COLOR_RED);
    init_pair(4, COLOR_YELLOW, COLOR_BLUE);
    init_pair(5, COLOR_BLACK, COLOR_WHITE);
    init_pair(6, COLOR_MAGENTA, COLOR_BLUE);
    init_pair(7, COLOR_CYAN, COLOR_WHITE);

    for (i = 1; i <= 7; i++) {
        attroff(A_BOLD);
        attrset(COLOR_PAIR(i));
        mvprintw(5 + i, 5, "Color pair %d", i);
        attrset(COLOR_PAIR(i) | A_BOLD);
```

6

```
            mvprintw(5 + i, 25, "Bold color pair %d", i);
            refresh();
            sleep(1);
        }

        endwin();
        exit(0);
    }
```

次の図はプログラムの実行例です。

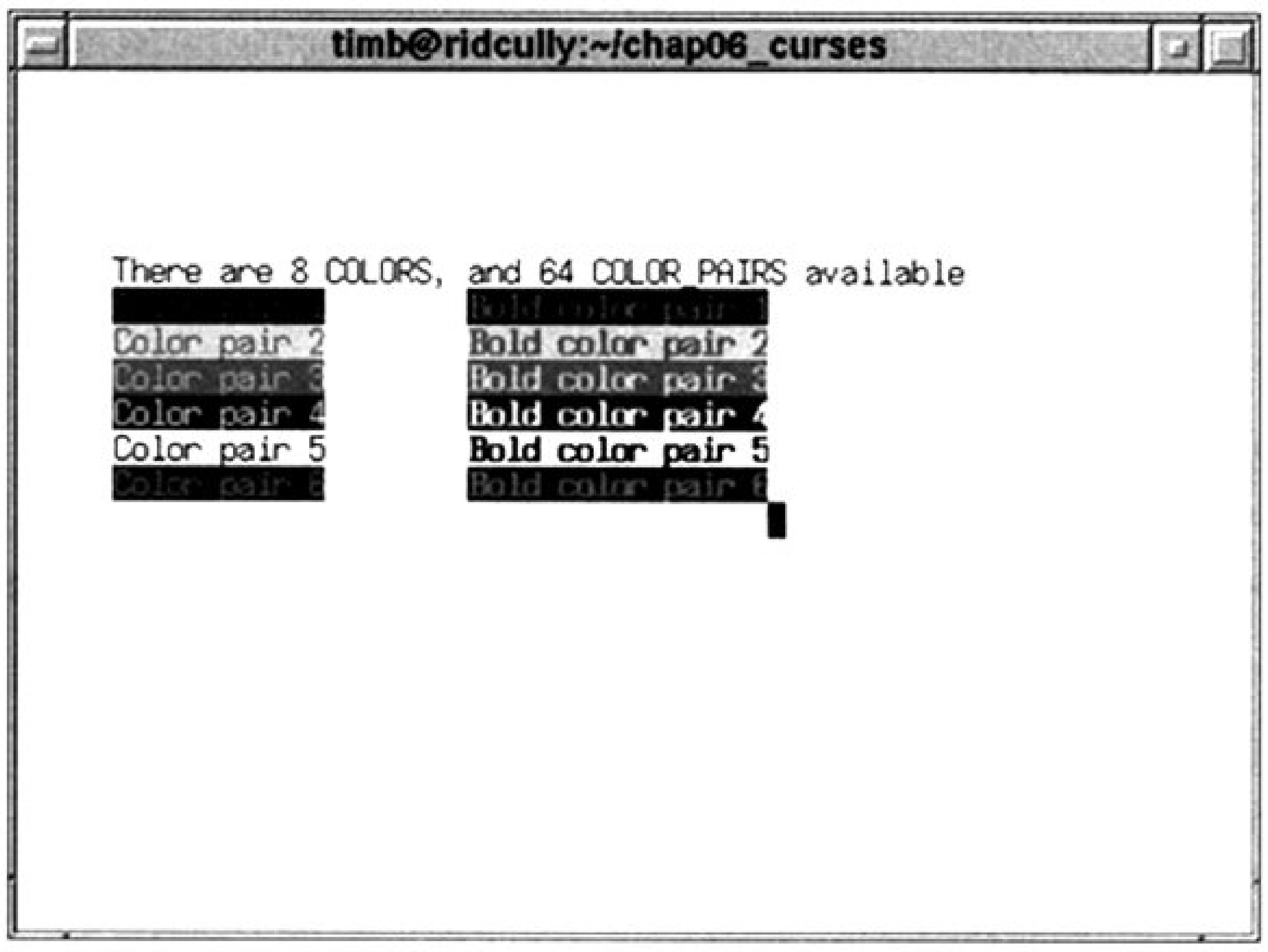


図 6.6 カラー

6.8.1 カラーの再定義

端末の中には、画面に同時に表示できるカラー数は制限されているものの、利用可能なカラーを再定義できるものがあります。**curses**では、**init_color**関数によってこのような機能をサポートしています。

```
#include <curses.h>

int init_color(short color_number, short red, short green, short blue);
```

init_color関数を使うと、0～1000の範囲の新しい輝度の値を使って、0から**COLORS**までの範囲の既存のカラーを再定義することができます。具体的な指定方法は、PCの画面でVGAパレットのカラーの値を定義する方法とほぼ同じです。

6.9 パッド

cursesを使って高機能なプログラムを作成する場合には、論理画面を用意しておき、その一部または全部をあとで物理画面に出力すると、プログラムを簡潔に記述できることがあります。また、物理画面より大きな論理画面を作成し、論理画面の一部だけを表示できると便利な場合もあります。

これまでに説明した**curses**の関数でこのような処理を実現することは困難です。作成するウィンドウは、いずれも物理画面の内側に収まっていなければならないからです。しかし、**curses**では、通常のウィンドウには収まらない論理画面情報を操作するための特別なデータ構造として**パッド**というものが用意されています。

パッド構造体は**WINDOW**構造体によく似ており、**WINDOW**に書き込む**curses**のすべてのルーチンは、パッドに対しても使用できます。ただし、パッドの作成と更新には独自のルーチンを使います。

パッドは、通常のウィンドウとほぼ同じ方法で作成することができます。

```
#include <curses.h>

WINDOW *newpad(int number_of_lines, int number_of_columns);
```

戻り値が、**newwin**と同じく**WINDOW**構造体へのポインタであることに注意してください。パッドを削除するには、通常のウィンドウの場合と同様、**delwin**を使います。

パッドの更新用のルーチンは、通常のウィンドウのものとは異なります。パッドは特定の画面領域の中に収まるものではないので、パッドのどの領域を画面上に配置するかだけでなく、パッドを画面上のどの位置に置くかについても指定する必要があります。パッドを更新するには、**prefresh**関数を使います。

```
#include <curses.h>

int prefresh(WINDOW *pad_ptr, int pad_row, int pad_column,
             int screen_row_min, int screen_col_min,
             int screen_row_max, int screen_col_max);
```

prefresh関数は、**(pad_row, pad_column)**を起点するパッドの領域を、**(screen_row_min, screen_col_min)**から**(screen_row_max, screen_col_max)**までの領域の画面に書き込みます。

パッドでは、効率的に画面の更新を行うための**pnoutrefresh**関数も用意されています。**pnoutrefresh**関数は、すでに説明した**wnoutrefresh**と同じように動作します。

次の**pad.c**は、パッドを使ったサンプルプログラムです。

例題 パッドの使用

1 プログラムの最初でパッド構造体を初期化し、パッドを作成してポインタを受け取ります。作成したパッド構造体は、行、カラムともに端末より50文字だけ大きいものです。このパッド構造体に文字を追加します。

```
#include <unistd.h>
#include <curses.h>

int main()
{
    WINDOW *pad_ptr;
    int x, y;
    int pad_lines;
    int pad_cols;
    char disp_char;

    initscr();

    pad_lines = LINES + 50;
    pad_cols = COLS + 50;

    pad_ptr = newpad(pad_lines, pad_cols);

    disp_char = 'a';
    for (x = 0; x < pad_lines; x++) {
        for (y = 0; y < pad_cols; y++) {
            mvwaddch(pad_ptr, x, y, disp_char);
            if (disp_char == 'z') disp_char = 'a';
            else disp_char++;
        }
    }
```

2 画面上のいろいろな位置にパッドの一部を描画します。

```
    prefresh(pad_ptr, 5, 7, 2, 2, 9, 9);
    sleep(1);

    prefresh(pad_ptr, LINES + 5, COLS + 7, 5, 5, 21, 19);
    sleep(1);

    delwin(pad_ptr);

    endwin();
    exit(0);
}
```

次の図はプログラムの実行例です。



図 6.7　パッドの使用

6.10 CDデータベースアプリケーション

以上で、**curses**で利用できる機能についてひととおり説明を終えました。ここでは、第2章の最後で作成したCDデータベースアプリケーションを、**curses**ライブラリを使ってCで書き直します。修正後のプログラムでは、画面に表示される情報が見やすくなっています。また、スクロールウィンドウを使ってトラックリストを表示します。

アプリケーション全体のコードは数ページにわたる長いものなので、以下ではセクションごと、関数ごとにコードを示します。フルソースコードは、ソフトバンクのWebサイト（**http://www.softbank.co.jp/books/editor/1st/linux/**）で入手することができます。本書のほかのプログラム同様、CDデータベースアプリケーションもGNU General Public Licenseに従って自由に改変、配布することができます。

メモ

Cバージョンのデータベースアプリケーションは、第5章と第6章で学んだことを応用して作成されています。特にCで実装することを念頭において設計し直したものではないので、シェルバージョンのプログラムの機能がほぼそのままの形で残されています。
プログラムにはいくつか問題もあります。たとえば、カンマを含むタイトルを正しく処理できません。また、画面表示の関係から、1枚のCDあたりのトラック数についても実質的に制限があります。これらの問題については、あとの章で解決策を示します。

コードはいくつかのセクションに分割できるので、以下ではセクションごとに見出しを付けて

います。また、関数の定義や、関数が呼び出されている箇所を強調表示しています。

例題 新しいCDデータベースアプリケーション

1 まず、必要なヘッダーファイルをインクルードし、グローバル定数を定義します。

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <curses.h>

#define MAX_STRING (80)         /* Longest allowed response       */
#define MAX_ENTRY (1024)        /* Longest allowed database entry */

#define MESSAGE_LINE 6          /* Misc. messages on this line    */
#define ERROR_LINE   22         /* Line to use for errors         */
#define Q_LINE       20         /* Line for questions             */
#define PROMPT_LINE  18         /* Line for prompting on          */
```

2 次に、グローバル変数を用意します。変数`current_cd`は、現在の作業対象となるCDタイトルを格納するのに使います。この変数は、先頭の文字を`\0`として初期化しておきます。これで、プログラムの実行開始直後には、選択されているCDタイトルがないことになります。`\0`は厳密には不要ですが、このように記述しておくことで変数を確実に初期化できます。もうひとつの変数`current_cat`は、現在のCDのカタログ番号を記録するのに使います。

```
static char current_cd[MAX_STRING] = "\0";
static char current_cat[MAX_STRING];
```

3 使用するいくつかのファイルの名前を宣言します。このバージョンでは、コードを簡潔にするために、一時ファイル名も含めてすべてのファイル名を固定しています。ただし、このような方法では、2人のユーザーが同じディレクトリでプログラムを実行した場合に問題が発生する可能性があります。

> **メモ**
> 本来なら、引数や環境変数を使ってデータベースファイルの名前を取得するほうがよいでしょう。また、一時ファイルについては、POSIXの`tmpnam`関数を使うなど、一意の名前を生成する方法を用意する必要があります。これらの点については、以降の章で解決することにします。

```
const char *title_file = "title.cdb";
const char *tracks_file = "tracks.cdb";
const char *temp_file = "cdb.tmp";
```

4 関数のプロトタイプ宣言を記述します。

```
void clear_all_screen(void);
void get_return(void);
int get_confirm(void);
int getchoice(char *greet, char *choices[]);
void draw_menu(char *options[], int highlight,
               int start_row, int start_col);
void insert_title(char *cdtitle);
void get_string(char *string);
void add_record(void);
void count_cds(void);
void find_cd(void);
void list_tracks(void);
void remove_tracks(void);
void remove_cd(void);
void update_cd(void);
```


5 関数の実装に進む前に、いくつかの構造体（実際にはメニューオプションの配列）を定義しておく必要があります。各オプションの最初の文字は、そのオプションが選択されたときに返す文字を表します。2文字目以降は、実際に画面に表示するテキストです。拡張メニューは、現在選択されているCDタイトルがあるときに表示されます。

```
char *main_menu[] =
{
    "aadd new CD",
    "ffind CD",
    "ccount CDs and tracks in the catalog",
    "qquit",
    0,
};

char *extended_menu[] =
{
    "aadd new CD",
    "ffind CD",
    "ccount CDs and tracks in the catalog",
    "llist tracks on current CD",
    "rremove current CD",
    "uupdate track information",
    "qquit",
    0,
};
```

以上でプログラムの初期化は終わりです。以下では、プログラムで使う各関数のコードを示します。関数は全部で15あり、相互の関係は次の図のようになっています。関数は次の3つのカテゴリに分類できます。

- メニューを表示する関数
- CDをデータベースに追加する関数
- CDデータを取得、表示する関数

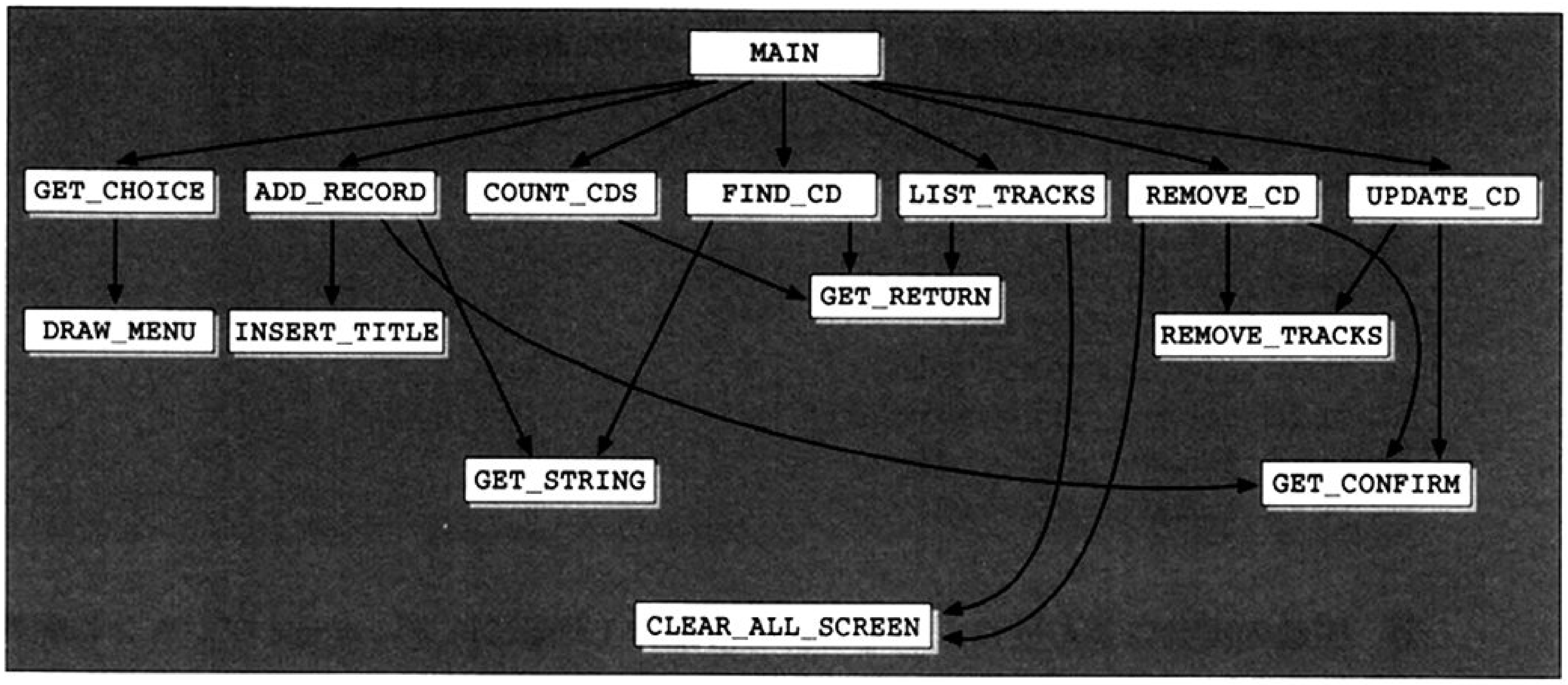


図 6.8 CDデータベースアプリケーションの関数

例題 main関数

1 main関数では、ユーザーがメニューから目的のオプションを選択します。ユーザーが**Q**を押した場合には、プログラムを終了します。

```
int main()
{
    int choice;
    initscr();
    do {
        choice = getchoice("Options:",
                           current_cd[0] ? extended_menu : main_menu);
        switch (choice) {
        case 'q':
            break;
        case 'a':
            add_record();
            break;
        case 'c':
            count_cds();
            break;
        case 'f':
            find_cd();
            break;
        case 'l':
            list_tracks();
            break;
```

```
            case 'r':
                remove_cd();
                break;
            case 'u':
                update_cd();
                break;
            }
        } while (choice != 'q');
        endwin();
        exit(EXIT_SUCCESS);
    }
```

次のセクションでは、プログラムのユーザーインタフェースを受け持つ3つの関数をまとめて示します。

例題 メニュー

1 **main**関数から呼び出される**getchoice**関数は、このセクションのかなめとなる関数です。**getchoice**は、簡単な説明を表す**greet**と、メニューへのポインタ**choices**を受け取ります。メニューへのポインタは、CDが選択されているかどうかに応じてメインメニューまたは拡張メニューのいずれかへのポインタになります（上の**main**関数を参照）。

```
int getchoice(char *greet, char *choices[])
{
    static int selected_row = 0;
    int max_row = 0;
    int start_screenrow = MESSAGE_LINE, start_screencol = 10;
    char **option;
    int selected;
    int key = 0;

    option = choices;
    while (*option) {
        max_row++;
        option++;
    }

    /* protect against menu getting shorter when CD deleted */
    if (selected_row >= max_row)
        selected_row = 0;

    clear_all_screen();
    mvprintw(start_screenrow - 2, start_screencol, greet);

    keypad(stdscr, TRUE);
    cbreak();
    noecho();

    key = 0;
    while (key != 'q' && key != KEY_ENTER && key != '\n') {
```

```
        if (key == KEY_UP) {
            if (selected_row == 0)
                selected_row = max_row - 1;
            else
                selected_row--;
        }
        if (key == KEY_DOWN) {
            if (selected_row == (max_row - 1))
                selected_row = 0;
            else
                selected_row++;
        }
        selected = *choices[selected_row];
        draw_menu(choices, selected_row, start_screenrow,
                              start_screencol);
        key = getch();
    }

    keypad(stdscr, FALSE);
    nocbreak();
    echo();

    if (key == 'q')
        selected = 'q';

    return (selected);
}
```

2 getchoiceで呼び出される関数には、clear_all_screenとdraw_menuがあります。次のコードは、draw_menu関数です。

```
void draw_menu(char *options[], int current_highlight,
               int start_row, int start_col)
{
    int current_row = 0;
    char **option_ptr;
    char *txt_ptr;

    option_ptr = options;
    while (*option_ptr) {
        if (current_row == current_highlight) {
            mvaddch(start_row + current_row, start_col - 3, ACS_BULLET);
            mvaddch(start_row + current_row, start_col + 40, ACS_BULLET);
        } else {
            mvaddch(start_row + current_row, start_col - 3, ' ');
            mvaddch(start_row + current_row, start_col + 40, ' ');
        }

        txt_ptr = options[current_row];
        txt_ptr++;
        mvprintw(start_row + current_row, start_col, "%s", txt_ptr);
        current_row++;
```

```
        option_ptr++;
    }

    mvprintw(start_row + current_row + 3, start_col,
             "Move highlight then press Return ");

    refresh();
}
```

3 次のコードは、**clear_all_screen**関数です。**clear_all_screen**関数は画面をクリアし、タイトルをもう一度書き込みます。これで、CDが選択されている場合にはCDの情報が表示されるようになります。

```
void clear_all_screen()
{
    clear();
    mvprintw(2, Q_LINE, "%s", "CD Database Application");
    if (current_cd[0]) {
        mvprintw(ERROR_LINE, 0, "Current CD: %s: %s\n",
                 current_cat, current_cd);
    }
    refresh();
}
```

次のセクションでは、CDデータベースに対する追加や更新を行う関数を示します。**main**から呼び出される関数には、**add_record**、**update_cd**、**remove_cd**があります。

例題 データベースファイルの操作

1 まず、新しいCDのレコードをデータベースに追加する関数です。

```
void add_record()
{
    char catalog_number[MAX_STRING];
    char cd_title[MAX_STRING];
    char cd_type[MAX_STRING];
    char cd_artist[MAX_STRING];
    char cd_entry[MAX_STRING];

    int screenrow = MESSAGE_LINE;
    int screencol = 10;

    clear_all_screen();
    mvprintw(screenrow, screencol, "Enter new CD details");
    screenrow += 2;

    mvprintw(screenrow, screencol, "Catalog Number: ");
    get_string(catalog_number);
    screenrow++;
```

```
    mvprintw(screenrow, screencol, "      CD Title: ");
    get_string(cd_title);
    screenrow++;

    mvprintw(screenrow, screencol, "       CD Type: ");
    get_string(cd_type);
    screenrow++;

    mvprintw(screenrow, screencol, "        Artist: ");
    get_string(cd_artist);
    screenrow++;

    mvprintw(PROMPT_LINE-2, 5, "About to add this new entry:");
    sprintf(cd_entry, "%s,%s,%s,%s",
            catalog_number, cd_title, cd_type, cd_artist);
    mvprintw(PROMPT_LINE, 5, "%s", cd_entry);
    refresh();

    move(PROMPT_LINE, 0);
    if (get_confirm()) {
        insert_title(cd_entry);
        strcpy(current_cd, cd_title);
        strcpy(current_cat, catalog_number);
    }
}
```

add_record関数では、**get_string**、**get_confirm**、および**insert_title**関数が呼び出されています。

2 **get_string**関数は、現在の画面位置で文字列を読み取ります。末尾の改行文字は削除します。

```
void get_string(char *string)
{
    int len;

    wgetnstr(stdscr, string, MAX_STRING);
    len = strlen(string);
    if (len > 0 && string[len - 1] == '\n')
        string[len - 1] = '\0';
}
```

3 **get_confirm**関数はユーザーの確認を求めます。ユーザーの入力文字列を読み取り、先頭の文字が**Y**か**y**かを調べます。これ以外の文字の場合には、ユーザーの確認を得られなかったものとして処理します。

```
int get_confirm()
{
    int confirmed = 0;
    char first_char;

    mvprintw(Q_LINE, 5, "Are you sure? ");
    clrtoeol();
    refresh();

    cbreak();
    first_char = getch();
    if (first_char == 'Y' || first_char == 'y') {
        confirmed = 1;
    }
    nocbreak();

    if (!confirmed) {
        mvprintw(Q_LINE, 1, "    Cancelled");
        clrtoeol();
        refresh();
        sleep(1);
    }
    return confirmed;
}
```

4 **insert_title**関数は、CDデータベースにタイトルを追加します。具体的には、タイトルファイルの最後にタイトル文字列を追加します。

```
void insert_title(char *cdtitle)
{
    FILE *fp = fopen(title_file, "a");
    if (!fp) {
        mvprintw(ERROR_LINE, 0, "cannot open CD titles database");
    } else {
        fprintf(fp, "%s\n", cdtitle);
        fclose(fp);
    }
}
```

5 **main**から呼び出されるその他のファイル操作関数についても、ここでまとめて示しておきます。**update_cd**関数では、スクロールするボックスタイプのサブウィンドウを使います。この関数ではいくつかの定数が必要になりますが、**list_tracks**関数でも同じ定数が必要になるので、次のようにグローバルな定数として定義します。

```
#define BOXED_LINES    11
#define BOXED_ROWS     60
#define BOX_LINE_POS   8
#define BOX_ROW_POS    2
```

update_cd関数では、現在のCDのトラックをユーザーが再入力することができます。具体的な処理としては、以前のトラックレコードを削除し、新しいデータの入力を求めます。

```
void update_cd()
{
    FILE *tracks_fp;
    char track_name[MAX_STRING];
    int len;
    int track = 1;
    int screen_line = 1;
    WINDOW *box_window_ptr;
    WINDOW *sub_window_ptr;

    clear_all_screen();
    mvprintw(PROMPT_LINE, 0, "Re-entering tracks for CD. ");
    if (!get_confirm())
        return;
    move(PROMPT_LINE, 0);
    clrtoeol();

    remove_tracks();

    mvprintw(MESSAGE_LINE, 0, "Enter a blank line to finish");

    tracks_fp = fopen(tracks_file, "a");
```

次のコードは、スクロールするボックスタイプのウィンドウでユーザーが情報を入力する部分です。まずサブウィンドウを用意し、ウィンドウに沿ってボックスを描きます。次に、このボックスタイプのウィンドウの内側に新しくスクロールするウィンドウを追加します。

```
    box_window_ptr = subwin(stdscr, BOXED_LINES + 2, BOXED_ROWS + 2,
                            BOX_LINE_POS - 1, BOX_ROW_POS - 1);
    if (!box_window_ptr)
        return;
    box(box_window_ptr, ACS_VLINE, ACS_HLINE);

    sub_window_ptr = subwin(stdscr, BOXED_LINES, BOXED_ROWS,
                            BOX_LINE_POS, BOX_ROW_POS);
    if (!sub_window_ptr)
        return;
    scrollok(sub_window_ptr, TRUE);
    werase(sub_window_ptr);
    touchwin(stdscr);

    do {
        mvwprintw(sub_window_ptr, screen_line++, BOX_ROW_POS + 2,
                  "Track %d: ", track);
        clrtoeol();
        refresh();
        wgetnstr(sub_window_ptr, track_name, MAX_STRING);
        len = strlen(track_name);
```

```
        if (len > 0 && track_name[len - 1] == '\n')
            track_name[len - 1] = '\0';

        if (*track_name)
            fprintf(tracks_fp, "%s,%d,%s\n",
                    current_cat, track, track_name);
        track++;
        if (screen_line > BOXED_LINES - 1) {
            /* time to start scrolling */
            scroll(sub_window_ptr);
            screen_line--;
        }
    } while (*track_name);
    delwin(sub_window_ptr);

    fclose(tracks_fp);
}
```


6 次の**remove_cd**も**main**から呼び出される関数です。

```
void remove_cd()
{
    FILE *titles_fp, *temp_fp;
    char entry[MAX_ENTRY];
    int cat_length;

    if (current_cd[0] == '\0')
        return;

    clear_all_screen();
    mvprintw(PROMPT_LINE, 0, "About to remove CD %s: %s. ",
             current_cat, current_cd);
    if (!get_confirm())
        return;

    cat_length = strlen(current_cat);

    /* Copy the titles file to a temporary, ignoring this CD */
    titles_fp = fopen(title_file, "r");
    temp_fp = fopen(temp_file, "w");

    while (fgets(entry, MAX_ENTRY, titles_fp)) {
        /* Compare catalog number and copy entry if no match */
        if (strncmp(current_cat, entry, cat_length) != 0)
            fputs(entry, temp_fp);
    }
    fclose(titles_fp);
    fclose(temp_fp);

    /* Delete the titles file, and rename the temporary file */
    unlink(title_file);
    rename(temp_file, title_file);
```

```
    /* Now do the same for the tracks file */
    remove_tracks();

    /* Reset current CD to 'None' */
    current_cd[0] = '\0';
}
```

7 最後は**remove_tracks**関数です。**remove_tracks**関数は現在のCDからトラックを削除します。この関数は、**update_cd**と**remove_cd**から呼び出されます。

```
void remove_tracks()
{
    FILE *tracks_fp, *temp_fp;
    char entry[MAX_ENTRY];
    int cat_length;

    if (current_cd[0] == '\0')
        return;

    cat_length = strlen(current_cat);

    tracks_fp = fopen(tracks_file, "r");
    temp_fp = fopen(temp_file, "w");

    while (fgets(entry, MAX_ENTRY, tracks_fp)) {
        /* Compare catalog number and copy entry if no match */
        if (strncmp(current_cat, entry, cat_length) != 0)
            fputs(entry, temp_fp);
    }
    fclose(tracks_fp);
    fclose(temp_fp);

    /* Delete the tracks file, and rename the temporary file */
    unlink(tracks_file);
    rename(temp_file, tracks_file);
}
```

例題 CDデータベースに対する問い合わせ

1 コレクションとしてなにかを収集する場合には、集めたアイテムが今どれくらいあるかがわかると便利です。次の**count_cds**関数は、データベースを走査し、登録されているタイトルとトラックの数を数えます。

```
void count_cds()
{
    FILE *titles_fp, *tracks_fp;
    char entry[MAX_ENTRY];
    int titles = 0;
    int tracks = 0;
```

```
    titles_fp = fopen(title_file, "r");
    if (titles_fp) {
        while (fgets(entry, MAX_ENTRY, titles_fp))
            titles++;
        fclose(titles_fp);
    }
    tracks_fp = fopen(tracks_file, "r");
    if (tracks_fp) {
        while (fgets(entry, MAX_ENTRY, tracks_fp))
            tracks++;
        fclose(tracks_fp);
    }
    mvprintw(ERROR_LINE, 0,
             "Database contains %d titles, with a total of %d tracks.",
             titles, tracks);
    get_return();
}
```


2 CDのジャケットをなくした場合でも、データベースに情報を入力していれば、次の**find_cd**関数を使ってトラック情報を取得することができます。**find_cd**関数は、データベースで検索する部分文字列を受け取り、見つかったCDタイトルをグローバル変数**current_cd**にセットします。

```
void find_cd()
{
    char match[MAX_STRING], entry[MAX_ENTRY];
    FILE *titles_fp;
    int count = 0;
    char *found, *title, *catalog;

    mvprintw(Q_LINE, 0, "Enter a string to search for in CD titles: ");
    get_string(match);

    titles_fp = fopen(title_file, "r");
    if (titles_fp) {
        while (fgets(entry, MAX_ENTRY, titles_fp)) {

            /* Skip past catalog number */
            catalog = entry;
            if (found = strstr(catalog, ",")) {
                *found = '\0';
                title = found + 1;

                /* Zap the next comma in the entry to reduce it to
                   title only */
                if (found = strstr(title, ",")) {
                    *found = '\0';

                    /* Now see if the match substring is present */
                    if (found = strstr(title, match)) {
```

```
                        count++;
                        strcpy(current_cd, title);
                        strcpy(current_cat, catalog);
                    }
                }
            }
        }
        fclose(titles_fp);
    }
    if (count != 1) {
        if (count == 0) {
            mvprintw(ERROR_LINE, 0, "Sorry, no matching CD found. ");
        }
        if (count > 1) {
            mvprintw(ERROR_LINE, 0,
                     "Sorry, match is ambiguous: %d CDs found. ", count);
        }
        current_cd[0] = '\0';
        get_return();
    }
}
```

3 次の**list_tracks**関数は、選択されたCDのトラック情報を画面に表示します。この関数では、前の**update_cd**のところでサブウィンドウ用に定義した定数を使っています。

```
void list_tracks()
{
    FILE *tracks_fp;
    char entry[MAX_ENTRY];
    int cat_length;
    int lines_op = 0;
    WINDOW *track_pad_ptr;
    int tracks = 0;
    int key;
    int first_line = 0;

    if (current_cd[0] == '\0') {
        mvprintw(ERROR_LINE, 0, "You must select a CD first. ");
        get_return();
        return;
    }
    clear_all_screen();
    cat_length = strlen(current_cat);

    /* First count the number of tracks for the current CD */
    tracks_fp = fopen(tracks_file, "r");
    if (!tracks_fp)
        return;
    while (fgets(entry, MAX_ENTRY, tracks_fp)) {
        if (strncmp(current_cat, entry, cat_length) == 0)
            tracks++;
    }
```

```
fclose(tracks_fp);

 /* Make a new pad, ensure that even if there is only a single
   track the PAD is large enough so the later prefresh() is always
   valid.
 */
track_pad_ptr = newpad(tracks + 1 + BOXED_LINES, BOXED_ROWS + 1);
if (!track_pad_ptr)
    return;

tracks_fp = fopen(tracks_file, "r");
if (!tracks_fp)
    return;

mvprintw(4, 0, "CD Track Listing\n");

/* write the track information into the pad */
while (fgets(entry, MAX_ENTRY, tracks_fp)) {
    /* Compare catalog number and output rest of entry */
    if (strncmp(current_cat, entry, cat_length) == 0) {
        mvwprintw(track_pad_ptr, lines_op++, 0, "%s",
                  entry + cat_length + 1);
    }
}
fclose(tracks_fp);

if (lines_op > BOXED_LINES) {
    mvprintw(MESSAGE_LINE, 0,
             "Cursor keys to scroll, RETURN or q to exit");
} else {
    mvprintw(MESSAGE_LINE, 0, "RETURN or q to exit");
}
wrefresh(stdscr);
keypad(stdscr, TRUE);
cbreak();
noecho();

key = 0;
while (key != 'q' && key != KEY_ENTER && key != '\n') {
    if (key == KEY_UP) {
        if (first_line > 0)
            first_line--;
    }
    if (key == KEY_DOWN) {
        if (first_line + BOXED_LINES + 1 < tracks)
            first_line++;
    }
    /* now draw the appropriate part of the pad on the screen */
    prefresh(track_pad_ptr, first_line, 0,
             BOX_LINE_POS, BOX_ROW_POS,
             BOX_LINE_POS + BOXED_LINES, BOX_ROW_POS + BOXED_ROWS);
    key = getch();
}
```

6

```
    delwin(track_pad_ptr);
    keypad(stdscr, FALSE);
    nocbreak();
    echo();
}
```

4 次のコードは、上の2つの関数で使われている**get_return**関数です。**get_return**関数はキャリッジリターンを読み取ります。その他の文字はすべて無視します。

```
void get_return()
{
    int ch;

    mvprintw(ERROR_LINE+1, 0, "%s", " Press return ");
    refresh();
    while ((ch = getch()) != '\n' && ch != EOF)
        ;
}
```

次の図はプログラムの実行例です。

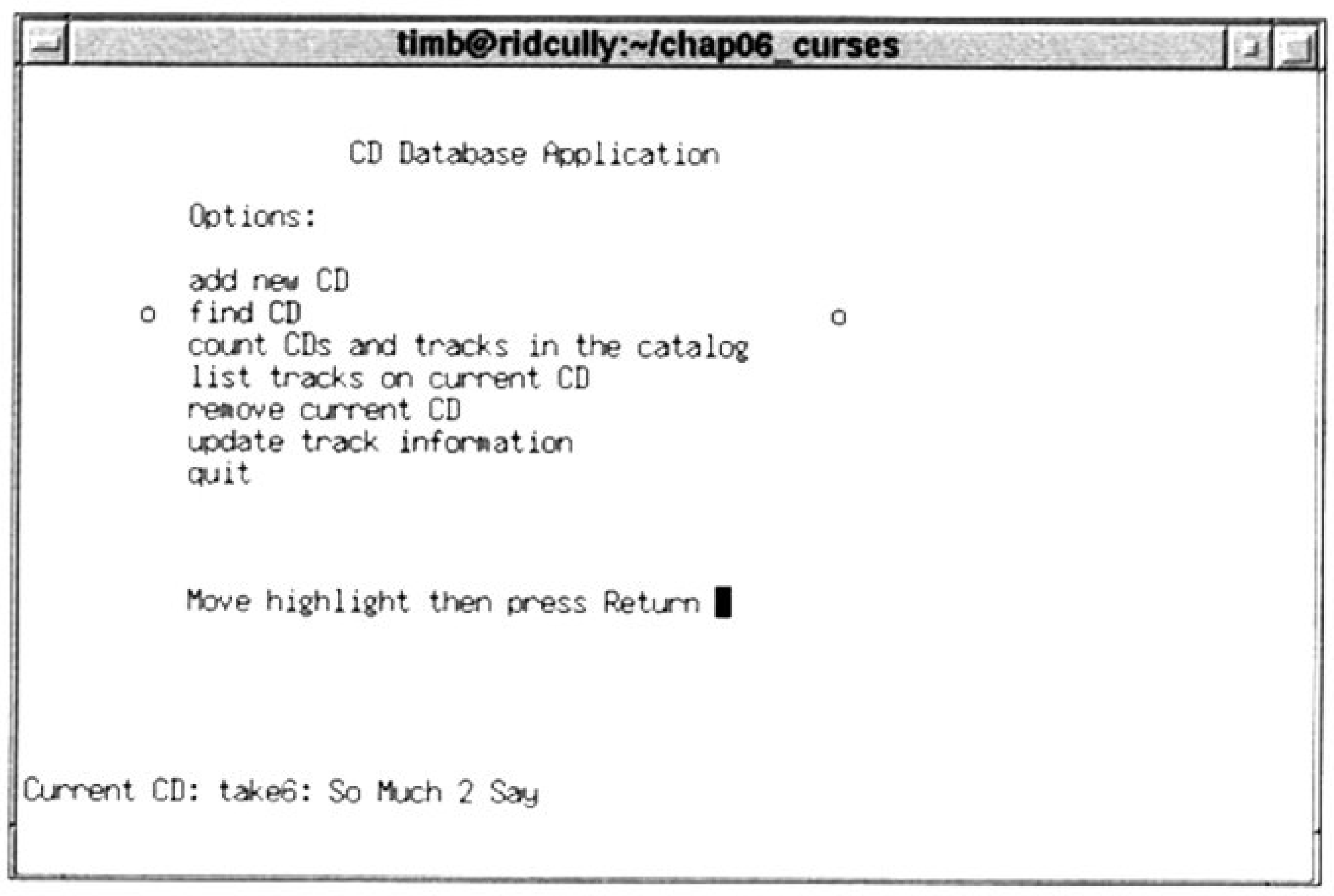


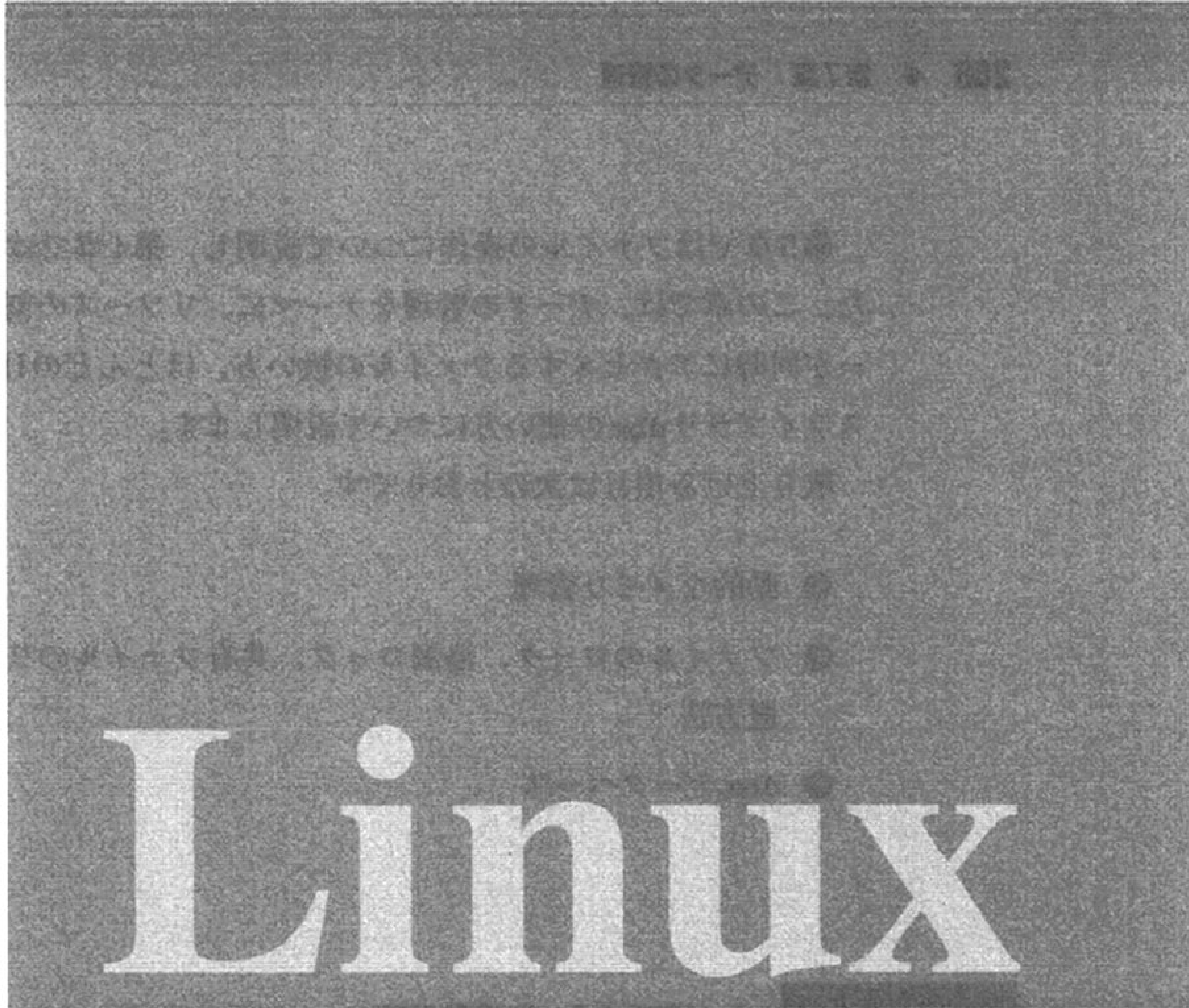
図 6.9 CDデータベースアプリケーション

6.11 この章のまとめ

この章では、cursesライブラリについて説明しました。cursesは、テキストベースのプログラムで画面を制御したりキーボードを読み取ったりするときに便利なライブラリです。

汎用端末インタフェース（GTI）と比べると、cursesライブラリでは端末を細かく制御したり、terminfoに直接アクセスしたりすることはできませんが、使い方ははるかに簡単です。フル画面のテキストベースアプリケーションを作成する場合には、cursesライブラリを使うと画面やキーボードを簡単に管理することができます。





第7章

データの管理

第3章ではファイルの操作について説明し、第4章ではリソースの制限について取り上げました。この章では、データの管理をテーマに、リソースの割り当てを管理する方法、複数のユーザーが同時にアクセスするファイルの扱い方、ほとんどのUNIXシステムで利用可能なデータベースライブラリ`dbm`の使い方について説明します。

取り上げる項目は次のとおりです。

- 動的なメモリ管理
- ファイルのロック、協調ロック、共有ファイルのリージョンのロック、デッドロックの回避方法
- `dbm`データベース

7.1 メモリの管理

どんなコンピュータでも、メモリは貴重なリソースです。また、どれだけたくさんのメモリが利用可能でも、決して十分とはいえないのがメモリです。1MBのメモリが必要になることはまずないだろうと思われていた時代もありましたが、今ではシングルユーザー用パーソナルコンピュータでさえ、最低8MBや16MBのメモリを搭載しています。

UNIXでは、最も初期のバージョンから、メモリ管理に関して非常に明解なアプローチをとっています。UNIXアプリケーションは、物理メモリに直接アクセスすることを許されていません。アプリケーションからは物理メモリにアクセスしているように見えますが、その実体は、いわばオペレーティングシステムが巧みに描き出した幻想です。

UNIXはプロセスに対して、フラットでセグメント化されていないメモリモデルを提供しています。各プロセスには、そのプロセス独自のメモリ領域が与えられます。また、ほとんどすべてのバージョンのUNIXがメモリ保護を提供しており、ほかのプロセスやオペレーティングシステムのメモリが、不正なプログラムによって上書きされないようになっています。一般に、あるプロセスに割り当てられたメモリは、ほかのどのプロセスからも読み書きすることはできません。UNIXでは、ハードウェアに用意されている機能を使って、こうしたプライベートなメモリの使い方を実現しています。

7.1.1 単純なメモリの割り当て

メモリを割り当てるには、標準Cライブラリの`malloc`を使います。

```
#include <stdlib.h>

void *malloc(size_t size);
```

一部のUNIXでは特別な**malloc.h**をインクルードしなければならないことがありますが、X/Open仕様ではその必要はありません。割り当てるバイト数を指定するパラメータ**size**は、符合なし整数型でなければなりませんが、単なる**int**ではなく、**size_t**である点に注意してください。

UNIXシステムでは非常に多くのメモリを割り当てることができます。さっそく試してみましょう。

例題 単純なメモリの割り当て

次のような内容の**memory1.c**プログラムを作成します。

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

#define A_MEGABYTE (1024 * 1024)

int main()
{
    char *some_memory;
    int  megabyte = A_MEGABYTE;
    int exit_code = EXIT_FAILURE;

    some_memory = (char *)malloc(megabyte);
    if (some_memory != NULL) {
        sprintf(some_memory, "Hello World\n");
        printf("%s", some_memory);
        exit_code = EXIT_SUCCESS;
    }
    exit(exit_code);
}
```

プログラムを実行すると、次のように出力されます。

```
$ memory1
Hello World
```

解説

このプログラムでは、**malloc**ライブラリを使って1MBのメモリへのポインタを受け取ります。**malloc**の呼び出しが成功したかどうかをチェックし、割り当てられたメモリの一部を使用して

該当するメモリが実際に存在することを確かめます。プログラムを実行すると、Hello World と表示され、割り当てられたメモリが存在していることがわかります。

mallocが返すポインタはvoid *型なので、結果をキャストしてchar *型にしています。malloc関数ではアラインされたメモリが返されることが保証されているので、戻り値を任意の型へのポインタにキャストすることができます。

このプログラムには、MS-DOSプログラムと大きく異なる点が2つあります。ひとつは、DOSエクステンダや込み入ったメモリモデルなどを使用せずに、1MBのメモリをひとつの大きなかたまりとして割り当てている点です。もうひとつは、単なる整数（int）を使って1MBという大きなメモリを割り当てている点です。通常のMS-DOSプログラミングモデルでは、intはたかだか16ビットです。したがって、符合なし整数を使ったとしても、16ビット幅整数で表現可能な最大値の65535バイトのメモリしか割り当てることができません。

一方、ほとんどのUNIXシステムは32ビット幅整数を使用し、メモリを指すのに32ビット幅のポインタを使用しているので、4GBまでのメモリを指定できます。つまり、UNIXでは、セグメントレジスタやその他のトリッキーなしくみを利用しなくても、32ビットポインタを使って広大なメモリ空間をダイレクトにアドレッシングすることが可能です。このようなメモリモデルのことを“**フラットな32ビットメモリモデル**”と呼びます。このメモリモデルは、32ビットアプリケーションを実行するWindows NTやWindows 95でも使われています。16ビットに制限されたUNIXも存在しますが、きわめてまれです。なお、64ビットバージョンのUNIXも存在するため、整数が32ビット幅であるという前提でプログラムを作成してはいけません。

7.1.2 大量のメモリの割り当て

UNIXには、DOSのメモリモデルの制限がありません。そこでもうひとつ、おもしろい実験をしてみましょう。筆者のマシンには16MBのメモリが搭載されていますが、このマシンでmallocを使ってメモリを割り当て続けるとどうなるでしょうか。カーネルや実行中のほかのプロセスがいくらかメモリを使っているはずですから、おそらく16MBよりやや少ないあたりでmallocの呼び出しが失敗すると考えられます。

例題 全物理メモリの割り当て要求

次のような内容のmemory2.cを作成し、マシンに搭載されているすべてのメモリを割り当ててみます。

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

#define A_MEGABYTE (1024 * 1024)
```

```
int main()
{
    char *some_memory;
    size_t  size_to_allocate = A_MEGABYTE;
    int  megs_obtained = 0;

    while (megs_obtained < 16) {
        some_memory = (char *)malloc(size_to_allocate);
        if (some_memory != NULL) {
            megs_obtained++;
            sprintf(some_memory, "Hello World");
            printf("%s - now allocated %d Megabytes\n", some_memory,
megs_obtained);
        }
        else {
            exit(EXIT_FAILURE);
        }
    }
    exit(EXIT_SUCCESS);
}
```


プログラムを実行すると、次のように出力されます（一部出力を省略しています）。

```
$ memory2
Hello World - now allocated 1 Megabytes
Hello World - now allocated 2 Megabytes
...
Hello World - now allocated 15 Megabytes
Hello World - now allocated 16 Megabytes
```

解説

最初のプログラムとほとんど同じですが、ループを使ってメモリの割り当てを繰り返しています。不思議なのは、物理メモリをすべて割り当てているはずなのに、プログラムがきちんと動作することです。なお、このプログラムでは、mallocを呼び出すときにsize_t型を使っています。現在のLinuxの実装では、これは実際にはunsigned intです。

timeコマンドを使ってプログラムの実行時間を測定してみましょう。

```
$ time memory2
```

筆者のシステムでは、1秒もかからずにプログラムが実行されました。つまり、すべてのメモリを割り当てることができただけでなく、非常に速く割り当てを行うことができたわけです。

それでは、いったいどれだけのメモリを割り当てることができるのでしょうか。次のサンプルプログラムmemory3.cでは、1KBずつメモリを割り当て、各メモリブロックに書き込みを行います。

例題 利用可能なメモリ

このサンプルプログラムmemory3.cは、システムを酷使するプログラムです。複数のユーザーでマシンを使っている場合には、ほかのユーザーに迷惑をかける可能性があるので、実行せずにコードを目で追うだけにとどめたほうがよいでしょう。

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

#define ONE_K (1024)

int main()
{
    char *some_memory;
    int  size_to_allocate = ONE_K;
    int  megs_obtained = 0;
    int  ks_obtained = 0;

    while (1) {
        for (ks_obtained = 0; ks_obtained < 1024; ks_obtained++) {
            some_memory = (char *)malloc(size_to_allocate);
            if (some_memory == NULL) exit(EXIT_FAILURE);
            sprintf(some_memory, "Hello World");
        }
        megs_obtained++;
        printf("Now allocated %d Megabytes\n", megs_obtained);
    }
    exit(EXIT_SUCCESS);
}
```

筆者のマシンでプログラムを実行すると、次のように出力されます（一部出力を省略しています）。

```
$ memory3
Now allocated 1 Megabytes
...
Now allocated 41 Megabytes
Now allocated 42 Megabytes
```

実行にはやや長い時間がかかり、12MBを過ぎたあたりから目に見えて動作が遅くなりました。

解説

アプリケーションに割り当てられるメモリは、UNIXカーネルが管理しています。UNIXカーネルは、プログラムがメモリを要求したり、割り当てられたメモリに対する読み取りや書き込みを行ったりするたびに、プログラムからの要求の処理方法を決定します。

UNIXカーネルは、最初のうちは、空いている物理メモリを使ってアプリケーションのメモリ要求を満たすことができます。しかし、いったん物理メモリに空きがなくなると、**スワップ領域**を使い始めます。スワップ領域は、ほとんどのバージョンのUNIXでは独立したディスク領域を意味し、Microsoft Windowsのスワップファイルと同じような働きをします。ただし、Microsoft Windowsの場合とは異なり、ローカルヒープやグローバルヒープ、あるいは廃棄可能なメモリセグメントについて考慮する必要はありません。代わりにUNIXカーネルがすべてを管理してくれます。

カーネルは、データとプログラムコードを物理メモリからスワップ領域へ、またスワップ領域から物理メモリへと移動します。このため、メモリが読み書きされる時点では、該当するデータがそれまで実際にどこに置かれていたかには関係なく、常に物理メモリ上に存在していたかのように見えます。

技術的な解説をすると、UNIXは"デマンドページド仮想メモリシステム"を実装しています。ユーザーから見えるすべてのメモリは**仮想メモリ**（バーチャルメモリ）であり、プログラムが使っている物理アドレスに実際に存在しているわけではありません。UNIXでは、すべてのメモリは**ページ**に分割されます。一般に、1ページは4096バイトです。プログラムがメモリにアクセスを試みると、仮想メモリから物理メモリへの変換が行われます。変換の方法、および変換にかかる時間は、使用しているハードウェアによって異なります。アクセス先のメモリが、物理的には存在していないメモリである場合には**ページフォールト**が発生し、制御はUNIXカーネルに移されます。

UNIXはアクセスされるアドレスをチェックしており、アクセス先のメモリがプログラムで使用できるアドレスである場合には、物理メモリのうちどのページを利用可能にするかを決定します。そして、該当するページがまだ書き込みの行われていないページである場合には、そのページを割り当てます。スワップ領域のディスクに格納されているページである場合には、目的のデータが含まれているメモリページを物理メモリに読み込みます（このとき、既存のページがディスクに追い出されることもあります）。次に、仮想メモリアドレスを物理メモリにマッピングし、ユーザープログラムに処理の続行を許可します。UNIXアプリケーションの側では、こうした操作についていっさい考慮する必要はありません。これらの操作はすべて、UNIXカーネル内部に隠された形で実装されているからです。

アプリケーションが物理メモリとスワップ領域の両方を使い果たしてしまった場合、または最大スタックサイズの制限を超えた場合、UNIXカーネルはそれ以上のメモリ要求を拒否します。

これまでに説明したことをアプリケーションプログラマの側から見ると、よいことずくめです。UNIXのメモリ管理は非常に優れており、単に大量のメモリを使用できるだけでなく、非常に多くのメモリを1つの**ブロック**として使用することができます。ひとつ注意が必要なのは、2つのメモリブロックを割り当てた場合、これらのメモリブロックが、連続してアドレッシング可能な単一のメモリブロックを形成するわけではないことです。メモリブロックを2つ要求した場合には、要求どおりに2つの独立したメモリブロックが割り当てられます。

さて、メモリは一見無制限に提供されているように見えますが、`malloc`からの戻り値をチェ

ックする意味はないのでしょうか。そんなことはありません。動的に割り当てられるメモリを使うCプログラムでよく問題になることとして、割り当てられたブロックを超えた部分への書き込みがあります。このようなことが行われた場合、プログラムはすぐに終了するわけではありませんが、mallocライブラリルーチンが内部的に使用しているデータを誤って上書きしている可能性があります。

通常、こうしたケースでは、次のmalloc呼び出しが失敗します。これは割り当てるメモリが不足しているからではなく、メモリ構造が破壊されてしまっているからです。こうした問題は追跡が困難であることが多く、エラーを検出したらなるべく早く原因を特定することが大切です。第9章ではデバッグと最適化について取り上げ、こうしたメモリ関連の問題を解決するのに役立つツールを紹介します。

7.1.3 メモリの不正使用

メモリを不正に使った場合には、どのようなことが起こるのでしょうか。次のmemory4.cでは、割り当てたメモリを超えた部分に書き込みを行ってプログラムの動作を確かめます。

例題 メモリの不正使用

```
#include <stdlib.h>

#define ONE_K (1024)

int main()
{
    char *some_memory;
    char *scan_ptr;

    some_memory = (char *)malloc(ONE_K);
    if (some_memory == NULL) exit(EXIT_FAILURE);

    scan_ptr = some_memory;
    while(1) {
        *scan_ptr = '\0';
        scan_ptr++;
    }
    exit(EXIT_SUCCESS);
}
```

プログラムを実行すると、次のように出力されます。

```
$ memory4
Segmentation fault
```

解説

UNIXのメモリ管理システムは、システムのほかの部分をメモリの不正使用から守っています。UNIXは、ほかのプログラムに悪影響が及ぶのを防ぐために、上のサンプルプログラムのような不正な動作をするプログラムを終了させます。

UNIXシステム上で実行される各プログラムから見えるメモリマップは、そのプログラム独自のものであり、ほかのすべてのプログラムのメモリマップとは異なります。物理メモリがどのように配置されているかはオペレーティングシステムだけが把握しており、オペレーティングシステムがユーザープログラムに代わってメモリを管理し、各ユーザープログラムをほかのユーザープログラムから保護しています。

7.1.4 ヌルポインタ

MS-DOSとは異なり、今日のUNIXシステムはヌルポインタに対する書き込みや読み取りを非常に厳しく制限しています（ただし、ヌルポインタへの書き込みや読み取りのときに実際にどのような動作をするかはシステムに依存します）。

例題 ヌルポインタへのアクセス

次の`memory5.c`は、ヌルポインタにアクセスしたときにどのようなことが起きるかを確認するプログラムです。

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main()
{
    char *some_memory = (char *)0;

    printf("A read from null %s\n", some_memory);
    sprintf(some_memory, "A write to null\n");
    exit(EXIT_SUCCESS);
}
```

プログラムを実行すると、次のように出力されます。

```
$ memory5
A read from null (null)
Segmentation fault
```

解説

printfでは、ヌルポインタが指し示す文字列の表示を試みています。次のsprintfでは、ヌルポインタへの書き込みを試みています。実行例からわかるように、Cライブラリはヌルポインタからの読み取りを許可しており、(n u l l) \0という文字列が返されています。一方、ヌルポインタへの書き込みでは、ただちにプログラムが終了させられています。ユーザープログラムから見えるすべてのメモリは仮想メモリなので、アドレス0は物理メモリのアドレス0を指すのではなく、プログラムが所有していないアドレス空間のページを指しています。

7.1.5 メモリの解放

さて、これまではメモリの割り当てだけを取り上げ、使用したメモリはプログラムの終了時に適切に処理されるものとして話を進めてきました。実際、UNIXのメモリ管理システムでは、プログラムの終了時にメモリがシステムに返されることを保証しています。しかし、単にメモリを割り当て、それを使ったあとに終了するだけのプログラムはまれです。むしろ実際のプログラムでは、必要に応じて動的にメモリを使うほうがふつうです。

動的にメモリを使うプログラムでは、使っていないメモリをfreeによってmallocメモリマネージャに返す必要があります。不要なメモリを解放すれば、分断されたメモリブロックはマージされ、アプリケーションに代わってmallocライブラリがメモリを管理してくれます。実行中のプログラム（プロセス）がメモリを使用したあとに解放した場合、解放されたメモリはプロセスに割り当てられたままです。ただし、そのメモリが使われていなければ、UNIXのメモリマネージャによって該当するメモリを物理メモリからスワップ領域にページアウトすることが可能になるので、リソースを有効に活用することができます。

```
#include <stdlib.h>

void free(void *ptr_to_memory);
```

freeの引数は、malloc、calloc、またはreallocを呼び出して割り当てられたメモリへのポインタでなければなりません。callocとreallocについてはすぐあとで取り上げます。

例題 メモリの解放

次のmemory6.cは、freeの使い方を示すプログラムです。

```
#include <stdlib.h>

#define ONE_K (1024)

int main()
```

```
{
    char *some_memory;
    int exit_code = EXIT_FAILURE;

    some_memory = (char *)malloc(ONE_K);
    if (some_memory != NULL) {
        free(some_memory);
        exit_code = EXIT_SUCCESS;
    }
    exit(exit_code);
}
```

解説

以前に割り当てられたメモリへのポインタを指定して`free`を呼び出しています。

> **注意**
> あるメモリブロックに対して`free`を呼び出すと、それ以後、そのメモリブロックは`malloc`ライブラリによって管理されます。`free`を呼び出して解放したメモリに対して読み取りや書き込みを行ってはいけません。

7.1.6　その他のメモリ割り当て関数

`malloc`や`free`ほどは使われませんが、メモリの割り当てを行う関数として`calloc`と`realloc`があります。

```
#include <stdlib.h>

void *calloc(size_t number_of_elements, size_t element_size);
void *realloc(void *existing_memory, size_t new_size);
```

`malloc`同様、`calloc`も`free`で解放できるメモリを割り当てますが、パラメータはやや異なります。`calloc`は、構造体の配列に使うメモリを割り当てるのに使用し、要素の数と各要素のサイズをパラメータとして指定します。割り当てられたメモリは0で埋められ、呼び出しが成功すると、最初の要素へのポインタが返されます。次に`calloc`を呼び出したときに、以前の空間と連続した空間が返されることが保証されない点は、`malloc`の場合と同じです。したがって、`calloc`を呼び出して作成された配列を、`calloc`をもう一度呼び出すことで拡張することはできません。

`realloc`関数は、以前に割り当てられたメモリブロックのサイズを変更します。`realloc`に渡すパラメータ`existing_memory`と`new_size`には、以前に`malloc`、`calloc`、または`realloc`によって割り当てられたメモリへのポインタと新しいサイズを指定します。`realloc`関数は、要求された操作を実行するためにデータを移動することがあります。したがって、

reallocを使ってメモリを割り当て直した場合には、reallocから返された新しいポインタを使う必要があります。reallocを呼び出す前に使っていた以前のポインタを使ってはいけません。

reallocを使う場合には、もうひとつ注意が必要です。reallocは、メモリのサイズを変更することができない場合にはヌルポインタを返します。つまり、次のようなコードを記述すると、問題が起きる可能性があります。

```
my_ptr = malloc(BLOCK_SIZE);
....
my_ptr = realloc(my_ptr, BLOCK_SIZE * 10);
```

もしreallocの呼び出しが失敗すると、my_ptrにはヌルポインタがセットされます。このため、最初にmallocで割り当てられたメモリにmy_ptrを使ってアクセスすることはできなくなります。このような問題を回避するには、reallocではなくmallocを使って新しいメモリを要求し、最初に確保したメモリブロックから新しいメモリブロックにmemcpyを使ってデータをコピーしたあと、freeを使って最初のメモリブロックを解放します。この方法を使えば、エラーが発生した場合でも最初のメモリブロックに格納されていたデータにはアクセスできるので、事後処理を適切に行ってからプログラムを終了することができます。

7.2 ファイルのロック

ファイルのロックは、マルチユーザー/マルチタスクオペレーティングシステムにとって非常に重要な意味を持っています。プログラムではしばしばデータを共有する必要があり、通常はファイルを通じてデータの共有を実現します。そのためには、プログラムがなんらかの方法によってファイルを自分の管理下に置く必要があります。ファイルを自分の管理下に置けば、安全に更新することができます。また、ほかのプログラムも、読み取りを行おうとしているファイルが、別のプログラムによる書き込み中などで過渡的な状態にある場合には、読み取り操作を中止することができます。

UNIXには、ファイルのロックに使用できる機能がいくつか用意されています。最も単純なテクニックとして、ロックの作成中に不適切な処理が行われないように、アトミックな操作でロックファイルを作成する方法があります。この方法では、一意であることが保証されるファイルを作成することができ、別のプログラムによって同時にファイルが作成されることもありません。

もうひとつ、より高機能なテクニックとして、ファイルの一部をロックして排他的にアクセスする方法があります。X/Openに準拠したバージョンのUNIXでは、この2番目のロックを実現するために、2つの異なる方法が用意されています。ここでは、これらの方法のうち一方だけを詳しく取り上げます。もうひとつの方法もしくみとしては非常によく似ており、プログラミングインタフェースが少し異なるだけです。

7.2.1 ロックファイルの作成

多くのアプリケーションでは、使用するリソースの**ロックファイル**を作成すれば十分です。この場合、ほかのプログラムは、ロックファイルを調べてリソースにアクセスできるかどうかを判断します。

一般に、この種のロックファイルは特別な場所に置かれ、制御対象のリソースに関連する名前が付けられます。たとえば、モデムを使う場合、Linuxでは`/usr/spool/uucp`ディレクトリにロックファイルを作成します（ディストリビューションによっては、別のディレクトリを使っていることがあります）。また、多くのUNIXシステムも、シリアルポートが使用できるかどうかを示すために`/usr/spool/uucp`ディレクトリを使っています。

```
$ ls /usr/spool/uucp
LCK..ttyS1
```


ロックファイルは、単にインジケータとしての役割を果たすだけです。つまり、ロックファイルの使用に関しては、さまざまなプログラムが互いに協調する必要があります。この種のロックは、（必須ロックと対比させて）**アドバイザリロック**（忠告ロック）と呼ばれます。

ロックインジケータとして使うファイルを作成するには、`O_CREAT`フラグと`O_EXCL`フラグを指定して`open`システムコールを呼び出します（`open`システムコールは`fcntl.h`で定義されており、本書ではすでに第3章で取り上げました）。この方法で`open`を呼び出すと、ファイルがまだ存在していないことを確認したうえで、単一のアトミックな操作でファイルを作成することができます。

例題 ロックファイルの作成

次の`lock1.c`は、ロックファイルを作成するプログラムです。

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>

int main()
{
    int file_desc;
    int save_errno;

    file_desc = open("/tmp/LCK.test", O_RDWR | O_CREAT | O_EXCL, 0444);
    if (file_desc == -1) {
        save_errno = errno;
        printf("Open failed with error %d\n", save_errno);
    }
```

```
    else {
        printf("Open succeeded\n");
    }
    exit(EXIT_SUCCESS);
}
```

プログラムを初めて実行した場合、出力は次のようになります。

```
$ lock1
Open succeeded
```

しかし、もう一度プログラムを実行すると、結果は次のようになります。

```
$ lock1
Open failed with error 17
```

解説

このプログラムでは、O_CREATフラグとO_EXCLフラグを指定してopenを呼び出し、/tmp/LCK.testという名前のファイルを作成しています。プログラムを初めて実行したときには、ファイルがまだ存在しておらず、openの呼び出しは成功します。しかし、それ以降はファイルがすでに存在するために、呼び出しは失敗します。プログラムが正常に終了するようにするには、ロックファイルを削除する必要があります。

Linuxシステムの場合、エラー番号17はEEXISTを意味し、ファイルがすでに存在していることを示します。エラー番号は、ヘッダーファイルerrno.h（またはこのファイルでインクルードされているファイル）で定義されています。次に示すのは、エラー番号17の定義です。

```
#define EEXIST          17      /* File exists */
```

これは、open(O_CREATE | O_EXCL)の失敗を表す適切なエラーです。

リソースへの排他的なアクセスが必要になるのがプログラムの実行中の短い期間（しばしば**クリティカルセクション**と呼ばれます）だけである場合には、クリティカルセクションに入る前にロックファイルを作成し、クリティカルセクションを出たらunlinkを使ってファイルを削除します。

このロック機構が複数のプログラムによってどのように利用されるかは、簡単なプログラムで確かめることができます。それには、同じプログラムを2つ同時に実行します。次のサンプルプログラムlock2.cでは、同時に実行する各プログラムを特定するために、第4章で取り上げたgetpidを使ってプロセスIDを取得します。

例題 協調ロックファイル

1 次に示すのは、ソースファイルの先頭からクリティカルセクションの前までです。

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>

const char *lock_file = "/tmp/LCK.test2";

int main()
{
    int file_desc;
    int tries = 10;

    while (tries--) {
        file_desc = open(lock_file, O_RDWR | O_CREAT | O_EXCL, 0444);
        if (file_desc == -1) {
            printf("%d - Lock already present\n", getpid());
            sleep(3);
        }
        else {
```

2 クリティカルセクションが始まります。

```
            printf("%d - I have exclusive access\n", getpid());
            sleep(1);
            (void)close(file_desc);
            (void)unlink(lock_file);
```

3 クリティカルセクションを終わります。

```
            sleep(2);
        }
    }
    exit(EXIT_SUCCESS);
}
```

プログラムを実行するには、まずロックファイルが存在しない状態にしておく必要があります。

```
$ rm -f /tmp/LCK.test2
```

次のコマンドを使って同じプログラムを2つ同時に実行します。

```
$ lock2 & lock2
```

次のように出力されます。

```
1284 - I have exclusive access
1283 - Lock already present
1283 - I have exclusive access
1284 - Lock already present
1284 - I have exclusive access
1283 - Lock already present
1283 - I have exclusive access
1284 - Lock already present
1284 - I have exclusive access
1283 - Lock already present
1283 - I have exclusive access
1284 - Lock already present
1284 - I have exclusive access
1283 - Lock already present
1283 - I have exclusive access
1284 - Lock already present
1284 - I have exclusive access
1283 - Lock already present
1283 - I have exclusive access
1284 - Lock already present
```

これは、同時に実行した同じプログラムが、互いにどのように協調して動作しているかを示しています。プロセスIDは実行環境によって異なる可能性がありますが、プログラムの動作自体は同じです。

解説

プログラムの動作をわかりやすくするために、**while**を使って10回ループしています。プログラムは、固有のロックファイル**/tmp/LCK.test2**を作成してクリティカルなリソースへのアクセスを試みます。ロックファイルがすでに存在するために試みが失敗した場合には、しばらく待ってからもう一度操作を試みます。成功した場合にはリソースにアクセスでき、上の手順で示したクリティカルセクションの部分で、排他アクセスを必要とする処理を行うことができます。

このサンプルプログラムはデモンストレーションを目的としているので、クリティカルセクションの部分では短時間待機するだけにしています。リソースの使用を終えたら、ロックファイルを削除してロックを解除します。ロックを解除したあとは、次にロックをかける前に別の処理を行うこともできます（サンプルプログラムでは**sleep**を呼び出しています）。ロックファイルはバイナリセマフォとして機能し、各プログラムからの「リソースは使用可能か」という問いに対して「はい」か「いいえ」のいずれかの答えを返します。

ここで重要なのは、ロックが協調的な方法で実現されている点です。つまり、このしくみが正しく動作するためには、すべてのプログラムを適切に記述しなければなりません。たとえば、ロックファイルの作成に失敗したときに、ロックファイルを削除して操作を再試行するように記述されたプログラムがあるとします。このプログラムでは、再試行の時点でロックファイルを作成

することができますが、もとのロックファイルを作成したプログラムでは、リソースへの排他アクセスが失われたことを知る方法がありません。

7.2.2 リージョンのロック

ロックファイルの作成は、シリアルポートなどのリソースに対する排他アクセスを制御する方法として便利です。しかし、サイズの大きな共有ファイルへのアクセスを制御する方法としてはあまり適切ではありません。たとえば、あるプログラムが作成した1つの巨大なファイルを、ほかの多数のプログラムから同時に更新するケースを考えてみてください。具体例としては、あるプログラムがなんらかのデータを長期間にわたって連続して記録し、このデータをほかのいくつかのプログラムから処理する場合を想定するとよいでしょう。記録されたデータを処理する側のプログラムでは、データを記録するプログラムが終了するのを待っているわけにはいきません。データを記録するプログラムは、ずっと実行されたままになっているからです。したがって、データを処理する側のプログラムでは、なんらかの協調的な方法を通じて同じファイルに同時にアクセスできるようにする必要があります。

このような場合に便利なのが、ファイルのリージョンのロックです。リージョンのロックでは、目的のファイルの特定の部分をロックし、ほかの部分は別のプログラムからもアクセスできるようにします。UNIXでは、このしくみを実現するために、少なくとも2つの方法が用意されています。ひとつは、**fcntl**システムコールを使う方法です。もうひとつは、**lockf**を使う方法です。ここでは、より一般的な**fcntl**インタフェースを最初に取り上げます。**lockf**を使う方法も、**fcntl**を使う方法とほとんど同じです。

fcntlについては、すでに第3章でも取り上げました。

```
#include <fcntl.h>

int fcntl(int fildes, int command, ...);
```

fcntlは、オープンされたファイルデスクリプタを対象として、パラメータ**command**に応じてさまざまな操作を実行します。このうち、ファイルのロックに関係するものには次の3つがあります。

- **F_GETLK**
- **F_SETLK**
- **F_SETLKW**

これらのパラメータを使う場合、3つ目のパラメータは**struct flock**へのポインタでなければなりません。したがって、プロトタイプは実際には次のようになります。

```
int fcntl(int fildes, int command, struct flock *flock_structure);
```

flock（ファイルロック）構造体はシステムに依存する構造体ですが、少なくとも次のメンバを持っています。

```
short l_type;
short l_whence;
off_t l_start;
off_t l_len;
pid_t l_pid;
```

メンバ**l_type**は、次のいずれかの値を取ります。これらの値は**fcntl.h**で定義されています。

表7.1 flock構造体のメンバl_typeの値

値	説明
F_RDLCK	共有ロック（または"読み取り"ロック）。複数の異なるプロセスが、ファイルの同じ領域（またはオーバーラップする複数の領域）に対して共有ロックを持つことができる。共有ロックを持つプロセスが1つでも存在する場合、どのプロセスもそのリージョンに対して排他ロックを取得することはできない。共有ロックを取得するには、ファイルが読み取りアクセスまたは読み書きアクセスでオープンされていなければならない。
F_UNLCK	アンロック。ロックを解除するのに使われる。
F_WRLCK	排他ロック（または"書き込み"ロック）。ファイルの特定のリージョンに対して排他ロックを持つことができるのは、1つのプロセスだけである。あるプロセスが排他ロックを持つと、ほかのプロセスはそのリージョンに対してどのような種類のロックも持つことができなくなる。排他ロックを取得するには、ファイルが書き込みアクセスまたは読み書きアクセスでオープンされていなければならない。

l_whence、**l_start**、**l_len**は、ファイルの**リージョン**（隣接するバイトの集合）を定義するためのメンバです。**l_whence**は、**SEEK_SET**、**SEEK_CUR**、**SEEK_END**のいずれかでなければなりません。これらは**unistd.h**で定義されており、それぞれファイルの先頭、現在位置、ファイルの最後に対応します。**l_whence**は、リージョンの最初のバイト位置を表す**l_start**の相対オフセットを表します。通常は**SEEK_SET**が使われるので、**l_start**はファイルの先頭からカウントされます。**l_len**は、リージョンに含まれるバイト数を表します。

パラメータ**l_pid**は、ロックを保持しているプロセスを報告するのに使われます。あとの**F_GETLK**の説明を参照してください。

ファイル内の各バイトに適用可能なロックは、ある時点では1種類だけであり、共有アクセス用にロックするか、排他アクセス用にロックするか、ロックを解除するかのいずれかになります。

fcntlに指定できるコマンドやオプションの組み合せはたくさんあるので、以下では1つずつ順番に見ていきます。

◆ F_GETLK コマンド

F_GETLKコマンドは、fildesが示すオープンされているファイルに関するロック情報を取得します。F_GETLKコマンドは、実際にファイルのロックを試みるわけではありません。このコマンドを使う場合、呼び出し側のプロセスでは、作成したいロックの種類に関する情報を渡します。F_GETLKコマンドを指定して呼び出したfcntlは、ロックの作成を妨げるような情報があれば、その情報を返します。

flock構造体で使われる値は次のとおりです。

表7.2 flock構造体の値（F_GETLKコマンド）

値	説明
l_type	共有（読み取り）ロックの場合はF_RDLCK、排他（書き込み）ロックの場合はF_WRLCK
l_whence	SEEK_SET、SEEK_CUR、SEEK_ENDのいずれか
l_start	ファイル内の目的のリージョンの先頭バイト
l_len	ファイル内の目的のリージョンに含まれるバイト数
l_pid	ロックを持つプロセスのID

プロセスでは、F_GETLKコマンドを使って、ファイルのリージョンに対するロックの現在の状態を調べることができます。それには、flock構造体に適切な値を設定し、プロセスで設定したいロックの種類と対象となるリージョンを指定します。呼び出しが成功すると、fcntlは-1以外の値を返します。すでにファイルにロックが設定されていて、次にロックが要求されてもロックを設定することができない場合、fcntlは適切な情報でflock構造体を上書きします。次のロック要求が成功する場合、flock構造体は（l_typeにF_UNLCKが設定される点を除いて）変更されません。F_GETLKコマンドが必要な情報を取得できなかった場合には-1が返されます。

F_GETLKコマンドが成功した場合（戻り値が-1以外の場合）、呼び出し側のアプリケーションではflock構造体の内容をチェックし、変更されていないかどうかを調べる必要があります。すでにロックが設定されている場合、メンバl_pidには、ロックしているプロセスのプロセスIDが設定されるので、flock構造体が変更されているかどうかを調べるには、このフィールドをチェックするとよいでしょう。

◆ F_SETLK コマンド

F_SETLKコマンドは、fildesが示すファイルの一部に対してロックの設定または解除を試みます。flock構造体で使われる値のうち、F_GETLKコマンドの場合と異なるものは次のとおりです。

表7.3 flock構造体の値（F_SETLKコマンド）

値	説明
l_type	共有（読み取り）ロックの場合はF_RDLCK、排他（書き込み）ロックの場合はF_WRLCK、リージョンに対するロックを解除する場合はF_UNLCK。
l_pid	使用しない。

ロックが成功すると、fcntlは -1以外の値を返します。失敗すると -1を返します。この関数は常にただちに戻ります。

◆ F_SETLKWコマンド

F_SETLKWコマンドは、ロックを取得できない場合には取得できるまで待機する点を除いて、F_SETLKコマンドと同じです。いったん待機を始めると、F_SETLKWコマンドは、ロックを取得できるか、またはシグナルが発生するまで戻りません。シグナルについては第10章で取り上げます。

プログラムがあるファイルに対して保持しているすべてのロックは、関連するファイルデスクリプタがクローズされたときに自動的に解除されます。また、プログラムが終了した場合にも同様の処理が行われます。

7.2.3 read／writeとロック

ファイルのリージョンに対するロックを使う場合には、高水準のfreadやfwriteではなく、低水準のreadとwriteを使うことが非常に重要です。これは、freadとfwriteでは、読み書きされるデータがライブラリ内部でバッファリングされるためです。たとえば、freadを呼び出してファイルの先頭から100バイトを読み取るとします。この場合、実際には（そしてほぼ確実に）100バイト以上のデータが読み込まれ、100バイト目以降のデータはライブラリ内部にバッファリングされることになります。このとき、もう一度freadを呼び出して次の100バイトを読み取ると、実際にはライブラリ内部にバッファリングされたデータが読み取られ、ファイルからさらにデータを読み取るための低水準のreadの呼び出しは行われない可能性があります。

このことがなぜ問題になるのでしょうか。同じファイルを更新する2つのプログラムを例にとって考えてみましょう。ファイルは200バイトのデータ（すべて0）から構成されているとします。最初のプログラムが先に実行を開始し、ファイルの先頭100バイトに対して書き込みロックを取得します。次にfreadを使って、ファイルの先頭100バイトのデータを読み込みます。第3章で説明したように、freadは一度にBUFSIZバイトまでを読み込むので、実際にはファイル全体がメモリに読み込まれます。しかし、プログラムに渡されるのは先頭の100バイトだけです。

ここで、2番目のプログラムが実行を開始します。2番目のプログラムは、ファイルの後半100バイトに対して書き込みロックを取得します。最初のプログラムがロックを設定しているのはフ

ァイルの先頭100バイトの部分だけなので、2番目のプログラムはロックの取得に成功します。2番目のプログラムは、ファイルの100バイト目から199バイト目までに2を書き込み、ファイルをクローズし、ロックを解除してプログラムの実行を終了します。この時点で、最初のプログラムはファイルの後半100バイトをロックし、**fread**を呼び出して該当するデータを読み込みます。しかし、このデータはバッファリングされているので、最初のプログラムから見えるデータは、実際にファイルに格納されている100バイトの2ではなく、100バイトの0です。つまり、最初のプログラムが操作の対象にしているデータと、実際のファイルに格納されているデータとの間に食い違いが生じてしまいます。低水準の**read**と**write**を使えばこのような問題は起きません。

やや説明が込み入っていたかもしれませんが、ファイルのロックは言葉で説明するより実際に使うほうが簡単です。さっそくファイルのロックを使ったサンプルプログラムを作成してみましょう。

ロックの動作を確かめるには、ロックを設定するプログラムと、ロックをテストするプログラムの2つのプログラムが必要です。最初に示す**lock3.c**は、ロックを行うプログラムのほうです。

例題 fcntlを使ったファイルのロック

1 ヘッダーファイルをインクルードし、変数を宣言します。

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>

const char *test_file = "/tmp/test_lock";

int main()
{
    int file_desc;
    int byte_count;
    char *byte_to_write = "A";
    struct flock region_1;
    struct flock region_2;
    int res;
```

2 ファイルデスクリプタをオープンします。

```
    file_desc = open(test_file, O_RDWR | O_CREAT, 0666);
    if (!file_desc) {
        fprintf(stderr, "Unable to open %s for read/write\n",
test_file);
        exit(EXIT_FAILURE);
    }
```

3 ファイルにデータを書き込みます。

```
for(byte_count = 0; byte_count < 100; byte_count++) {
    (void)write(file_desc, byte_to_write, 1);
}
```

4 リージョン1を設定します。種類は共有ロックで、位置は10～30バイト目です。

```
region_1.l_type = F_RDLCK;
region_1.l_whence = SEEK_SET;
region_1.l_start = 10;
region_1.l_len = 20;
```

5 リージョン2を設定します。種類は排他ロックで、位置は40～50バイト目です。

```
region_2.l_type = F_WRLCK;
region_2.l_whence = SEEK_SET;
region_2.l_start = 40;
region_2.l_len = 10;
```

6 ファイルをロックします。

```
printf("Process %d locking file\n", getpid());
res = fcntl(file_desc, F_SETLK, &region_1);
if (res == -1) fprintf(stderr, "Failed to lock region 1\n");
res = fcntl(file_desc, F_SETLK, &region_2);
if (res == -1) fprintf(stderr, "Failed to lock region 2\n");
```

7 しばらく待機します。

```
    sleep(60);

    printf("Process %d closing file\n", getpid());
    close(file_desc);
    exit(EXIT_SUCCESS);
}
```

解説

まず、ファイルを作成し、ファイルを読み書き用にオープンしてデータを書き込みます。次に、リージョンを2つ設定します。ひとつは10～30バイト目までで、共有（読み取り）ロックとします。もうひとつは40～50バイト目までで、排他（書き込み）ロックとします。`fcntl`を呼び出して2つのリージョンをロックし、しばらく待機したあと、ファイルをクローズしてプログラムを終了します。

プログラムが待機し始めた時点で、ロックの状態は図のようになっています。

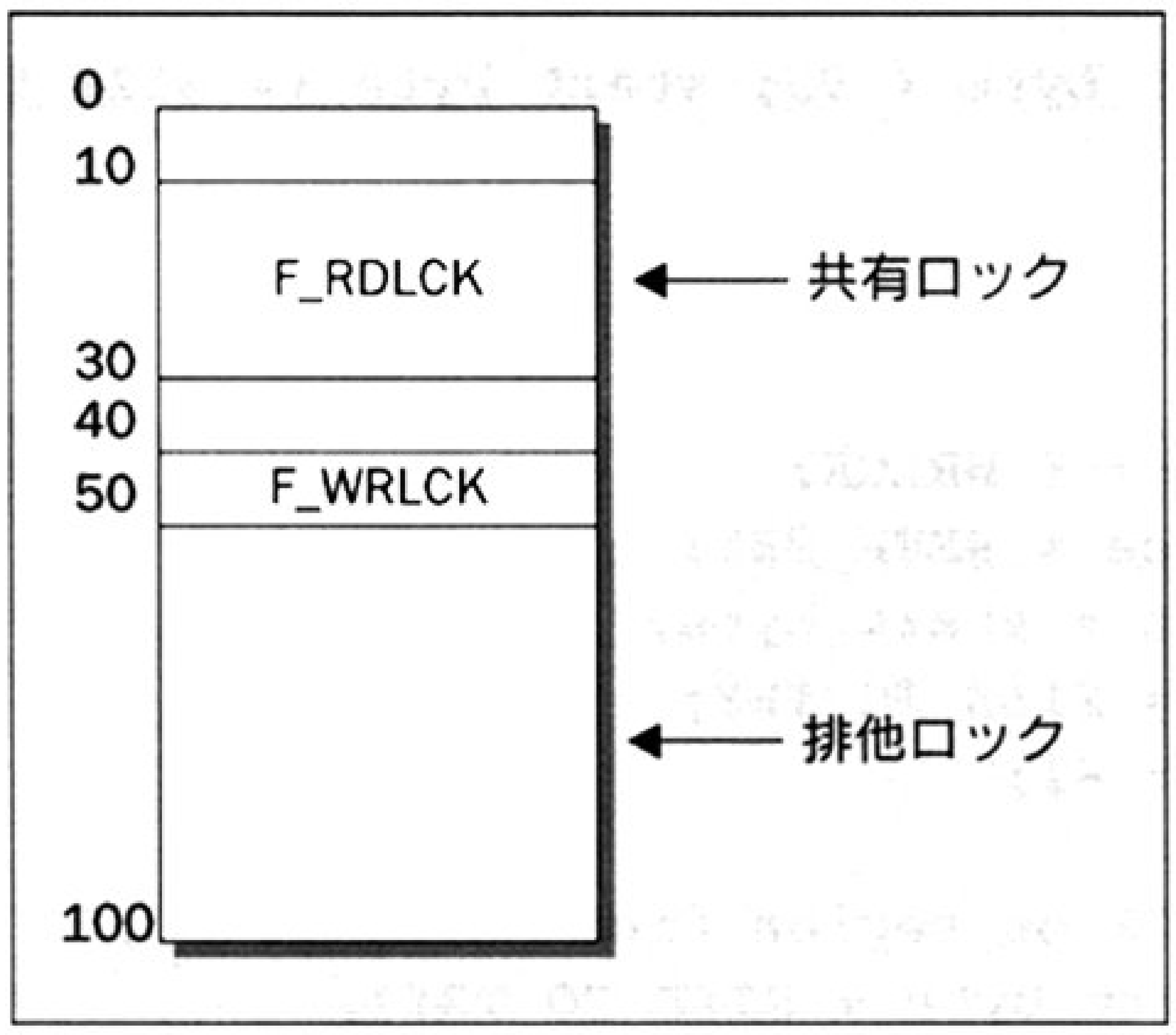


図7.1 ロックの状態

このプログラムは、それ自体ではあまり意味がありません。ロックをテストするには、もうひとつプログラムが必要です。ここでは、ロックをテストするプログラムを**lock4.c**という名前で作成します。

例題 ファイルのロックのテスト

1 ヘッダーファイルをインクルードし、変数を宣言します。

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>

const char *test_file = "/tmp/test_lock";
#define SIZE_TO_TRY 5

void show_lock_info(struct flock *to_show);

int main()
{
    int file_desc;
    int res;
    struct flock region_to_test;
    int start_byte;
```

2 ファイルデスクリプタをオープンします。

```
    file_desc = open(test_file, O_RDWR | O_CREAT, 0666);
    if (!file_desc) {
        fprintf(stderr, "Unable to open %s for read/write", test_file);
```

```
        exit(EXIT_FAILURE);
    }

    for (start_byte = 0; start_byte < 99; start_byte += SIZE_TO_TRY) {
```

3 テストするリージョンを設定します。

```
        region_to_test.l_type = F_WRLCK;
        region_to_test.l_whence = SEEK_SET;
        region_to_test.l_start = start_byte;
        region_to_test.l_len = SIZE_TO_TRY;
        region_to_test.l_pid = -1;

        printf("Testing F_WRLCK on region from %d to %d\n",
               start_byte, start_byte + SIZE_TO_TRY);
```

4 ファイルに対するロックをテストします。

```
        res = fcntl(file_desc, F_GETLK, &region_to_test);
        if (res == -1) {
            fprintf(stderr, "F_GETLK failed\n");
            exit(EXIT_FAILURE);
        }
        if (region_to_test.l_pid != -1) {
            printf("Lock would fail. F_GETLK returned:\n");
            show_lock_info(&region_to_test);
        }
        else {
            printf("F_WRLCK - Lock would succeed\n");
        }
```

5 今度は共有（読み取り）ロックでテストを繰り返します。まず、テストするリージョンを設定します。

```
        region_to_test.l_type = F_RDLCK;
        region_to_test.l_whence = SEEK_SET;
        region_to_test.l_start = start_byte;
        region_to_test.l_len = SIZE_TO_TRY;
        region_to_test.l_pid = -1;

        printf("Testing F_RDLCK on region from %d to %d\n",
               start_byte, start_byte + SIZE_TO_TRY);
```

6 ファイルに対するロックをテストします。

```
        res = fcntl(file_desc, F_GETLK, &region_to_test);
        if (res == -1) {
            fprintf(stderr, "F_GETLK failed\n");
            exit(EXIT_FAILURE);
```

```
        }
        if (region_to_test.l_pid != -1) {
            printf("Lock would fail. F_GETLK returned:\n");
            show_lock_info(&region_to_test);
        }
        else {
            printf("F_RDLCK - Lock would succeed\n");
        }
    }
    close(file_desc);
    exit(EXIT_SUCCESS);
}

void show_lock_info(struct flock *to_show) {
    printf("\tl_type %d, ", to_show->l_type);
    printf("l_whence %d, ", to_show->l_whence);
    printf("l_start %d, ", (int)to_show->l_start);
    printf("l_len %d, ", (int)to_show->l_len);
    printf("l_pid %d\n", to_show->l_pid);
}
```


実際にロックをテストするには、まず**lock3**プログラムを実行し、次に**lock4**プログラムを実行してロックされたファイルをテストします。具体的には、**lock3**プログラムをバックグラウンドで実行します。

```
$ lock3 &
$ process 1534 locking file
```

lock3プログラムがバックグラウンドで実行され、コマンドプロンプトが戻ってきます。ここで、**lock4**プログラムを実行します。

```
lock4
```

出力は次のようになります（一部省略しています）。

```
Testing F_WRLOCK on region from 0 to 5
F_WRLCK - Lock would succeed
Testing F_RDLOCK on region from 0 to 5
F_RDLCK - Lock would succeed
...
Testing F_WRLOCK on region from 10 to 15
Lock would fail. F_GETLK returned:
        l_type 0, l_whence 0, l_start 10, l_len 20, l_pid 1534
Testing F_RDLOCK on region from 10 to 15
F_RDLCK - Lock would succeed
Testing F_WRLOCK on region from 15 to 20
Lock would fail. F_GETLK returned:
        l_type 0, l_whence 0, l_start 10, l_len 20, l_pid 1534
Testing F_RDLOCK on region from 15 to 20
```

```
F_RDLCK - Lock would succeed
...
Testing F_WRLOCK on region from 25 to 30
Lock would fail. F_GETLK returned:
        l_type 0, l_whence 0, l_start 10, l_len 20, l_pid 1534
Testing F_RDLOCK on region from 25 to 30
F_RDLCK - Lock would succeed
...
Testing F_WRLOCK on region from 40 to 45
Lock would fail. F_GETLK returned:
        l_type 1, l_whence 0, l_start 40, l_len 10, l_pid 1534
Testing F_RDLOCK on region from 40 to 45
Lock would fail. F_GETLK returned:
        l_type 1, l_whence 0, l_start 40, l_len 10, l_pid 1534
...
Testing F_RDLOCK on region from 95 to 100
F_RDLCK - Lock would succeed
```

解説

lock4は、ファイルを5バイト単位で区切り、ロックをテストするためのリージョンを設定します。次に、書き込みまたは読み取りのいずれかでリージョンをロックできるかどうかを調べます。返された情報は、ロック要求が失敗する原因となるリージョンのバイト数、およびファイルの先頭からのオフセットなどです。**flock**構造体の**l_pid**には、現在ファイルをロックしているプログラムのプロセスIDが返されるので、**fcntl**を呼び出す前に -1（プロセスIDとして無効な値）を設定し、呼び出しから戻ったときに値が変わっているかどうかを調べます。リージョンがロックされていない場合、**l_pid**の値は変化しません。

プログラムからの出力を解読するには、インクルードファイル**fcntl.h**（Linuxでは**/usr/include/asm**にあります）を参照する必要があります。このファイルを見ると、**l_type**が1の場合には**F_WRLCK**を、**l_type**が0の場合には**F_RDLCK**を表すことがわかります。つまり、返された情報の**l_type**が1の場合には、書き込みロックがすでに存在するためにロック要求が失敗し、**l_type**が0の場合には、読み取りロックがすでに存在するためにロック要求が失敗します。**lock3**プログラムがロックしていないリージョンに対しては、共有ロックと排他ロックの両方が成功します。

10～30バイト目の部分では、共有ロックが可能であることがわかります。これは、**lock3**プログラムが設定しているロックが共有ロックであり、排他ロックではないからです。一方、40～50バイト目の部分では、共有ロックと排他ロックのいずれの要求も失敗します。これは、該当するリージョンに対して**lock3**プログラムが排他（**F_WRLCK**）ロックを持っているためです。

7.2.4 ロックの取り合い

ファイルに設定された既存のロックをテストする方法についてはすでに示したので、今度は1つのファイルの同じ部分に対して2つのプログラムがロックを取り合った場合にどうなるかを見ておきましょう。まず、前に作成したlock3プログラムを使ってファイルをロックします。次に、新しく作成するプログラムを使って同じファイルのロックを試みます。新しいプログラムでは、ロックを解除する呼び出しも使います。プログラムの名前はlock5.cとします。

例題 ロックの取り合い

1 ヘッダーファイルをインクルードして変数を宣言したあと、ファイルデスクリプタをオープンします。

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>

const char *test_file = "/tmp/test_lock";

int main()
{
    int file_desc;
    struct flock region_to_lock;
    int res;

    file_desc = open(test_file, O_RDWR | O_CREAT, 0666);
    if (!file_desc) {
        fprintf(stderr, "Unable to open %s for read/write\n", test_file);
        exit(EXIT_FAILURE);
    }
```

2 プログラムの残りの部分では、ファイル内のさまざまなリージョンを指定して異なる種類のロック操作を試みます。

```
    region_to_lock.l_type = F_RDLCK;
    region_to_lock.l_whence = SEEK_SET;
    region_to_lock.l_start = 10;
    region_to_lock.l_len = 5;
    printf("Process %d, trying F_RDLCK, region %d to %d\n", getpid(),
           (int)region_to_lock.l_start, (int)(region_to_lock.l_start +
           region_to_lock.l_len));
    res = fcntl(file_desc, F_SETLK, &region_to_lock);
    if (res == -1) {
        printf("Process %d - failed to lock region\n", getpid());
    } else {
```

7

```
        printf("Process %d - obtained lock region\n", getpid());
    }

    region_to_lock.l_type = F_UNLCK;
    region_to_lock.l_whence = SEEK_SET;
    region_to_lock.l_start = 10;
    region_to_lock.l_len = 5;
    printf("Process %d, trying F_UNLCK, region %d to %d\n", getpid(),
           (int)region_to_lock.l_start, (int)(region_to_lock.l_start +
           region_to_lock.l_len));
    res = fcntl(file_desc, F_SETLK, &region_to_lock);
    if (res == -1) {
        printf("Process %d - failed to unlock region\n", getpid());
    } else {
        printf("Process %d - unlocked region\n", getpid());
    }

    region_to_lock.l_type = F_UNLCK;
    region_to_lock.l_whence = SEEK_SET;
    region_to_lock.l_start = 0;
    region_to_lock.l_len = 50;
    printf("Process %d, trying F_UNLCK, region %d to %d\n", getpid(),
           (int)region_to_lock.l_start, (int)(region_to_lock.l_start +
           region_to_lock.l_len));
    res = fcntl(file_desc, F_SETLK, &region_to_lock);
    if (res == -1) {
        printf("Process %d - failed to unlock region\n", getpid());
    } else {
        printf("Process %d - unlocked region\n", getpid());
    }

    region_to_lock.l_type = F_WRLCK;
    region_to_lock.l_whence = SEEK_SET;
    region_to_lock.l_start = 16;
    region_to_lock.l_len = 5;
    printf("Process %d, trying F_WRLCK, region %d to %d\n", getpid(),
           (int)region_to_lock.l_start, (int)(region_to_lock.l_start +
           region_to_lock.l_len));
    res = fcntl(file_desc, F_SETLK, &region_to_lock);
    if (res == -1) {
        printf("Process %d - failed to lock region\n", getpid());
    } else {
        printf("Process %d - obtained lock on region\n", getpid());
    }

    region_to_lock.l_type = F_RDLCK;
    region_to_lock.l_whence = SEEK_SET;
    region_to_lock.l_start = 40;
    region_to_lock.l_len = 10;
    printf("Process %d, trying F_RDLCK, region %d to %d\n", getpid(),
           (int)region_to_lock.l_start, (int)(region_to_lock.l_start +
           region_to_lock.l_len));
    res = fcntl(file_desc, F_SETLK, &region_to_lock);
```

```
    if (res == -1) {
        printf("Process %d - failed to lock region\n", getpid());
    } else {
        printf("Process %d - obtained lock on region\n", getpid());
    }

    region_to_lock.l_type = F_WRLCK;
    region_to_lock.l_whence = SEEK_SET;
    region_to_lock.l_start = 16;
    region_to_lock.l_len = 5;
    printf("Process %d, trying F_WRLCK with wait, region %d to %d\n", getpid(),
           (int)region_to_lock.l_start, (int)(region_to_lock.l_start
           +region_to_lock.l_len));
    res = fcntl(file_desc, F_SETLKW, &region_to_lock);
    if (res == -1) {
        printf("Process %d - failed to lock region\n", getpid());
    } else {
        printf("Process %d - obtained lock on region\n", getpid());
    }

    printf("Process %d ending\n", getpid());
    close(file_desc);
    exit(EXIT_SUCCESS);
}
```

まず、lock3プログラムをバックグラウンドで実行したあと、新しいプログラムを実行します。出力は次のようになります。

```
Process 227 locking file
Process 228, trying F_RDLCK, region 10 to 15
Process 228 - obtained lock on region
Process 228, trying F_UNLCK, region 10 to 15
Process 228 - unlocked region
Process 228, trying F_UNLCK, region 0 to 50
Process 228 - unlocked region
Process 228, trying F_WRLCK, region 16 to 21
Process 228 - failed to lock on region
Process 228, trying F_RDLCK, region 40 to 50
Process 228 - failed to lock on region
Process 228, trying F_WRLCK with wait, region 16 to 21
Process 227 closing file
Process 228 - obtained lock on region
Process 228 ending
```

解説

最初に、10～15バイト目のリージョンに対して共有ロックの設定を試みます。このリージョンにはすでに共有ロックが設定されていますが、共有ロックは同時に複数設定することが許容されているので、ロックは成功します。

次に、同じリージョンに対する（自分が設定した）共有ロックを解除します。この操作も成功します。ファイルの先頭50バイトの部分にはプログラムが設定したロックは存在しませんが、このリージョンに対してロックの解除を試みます。これも成功します。これは、アンロック要求の処理後の状態と該当するリージョンに対してプログラムが保持しているロックが依存しない状態とが結果的にイコールになるからです。

続いて、16〜21バイト目のリージョンに対して排他ロックの設定を試みます。このリージョンにはすでに共有ロックが設定されているので、排他ロックを作成することができず、試みは失敗します。

その後、40〜50バイト目のリージョンに対して共有ロックの設定を試みます。このリージョンにはすでに排他ロックが設定されているので、試みは失敗します。

最後に、16〜21バイト目のリージョンに対してもう一度排他ロックの設定を試みます。ただし、今度は**F_SETLCKW**コマンドを使って、ロックを取得できるまで待機します。プログラムからの出力はしばらくの間止まります。該当するリージョンをロックしている**lock3**プログラムがファイルをクローズし、その結果、そのプログラムが取得していたロックがすべて解除されると、**lock5**プログラムは実行を再開し、リージョンのロックに成功します。

7.2.5 その他のロックコマンド

ファイルをロックするには、**fcntl**を使う方法のほかに**lockf**関数を使う方法もあります。

```
#include <unistd.h>

int lockf(int fildes, int function, off_t size_to_lock);
```

functionには次のいずれかの値を指定します。

- **F_ULOCK**　ロックの解除
- **F_LOCK**　排他ロック
- **F_TLOCK**　テストおよび排他ロック
- **F_TEST**　ほかのプロセスが設定したロックに関するテスト

パラメータ**size_to_lock**には、ロックの対象となるバイト数をファイル内の現在のオフセットを起点として指定します。

lockfは**fcntl**ほど柔軟ではなく、機能も制限されているので、インタフェースもそれだけシンプルです。**lockf**関数を使う場合には、ロックしたいリージョンの開始位置にシークし、ロックの対象となるバイト数を指定しなければなりません。

lockfで設定できるのは、**fcntl**の場合と同様、アドバイザリロックだけです。ロックによっ

てファイルに対する読み取りや書き込みが実際に禁止されるわけではありません。**fcntl**によるロックと**lockf**によるロックとを混在させて使った場合の結果は未定義です。したがって、いったんどちらかの方法を使うことを決めたら、一貫してその方法を使わなければなりません。

7.2.6 デッドロック

ロックを使う場合には、デッドロックの危険性についても考慮しなければなりません。たとえば、2つのプログラムが同じファイルを更新するとします。どちらのプログラムでも、対象となるファイルの1バイト目と2バイト目を同時に更新する必要があります。プログラムAでは、2バイト目を更新してから1バイト目を更新しますが、プログラムBでは、1バイト目を最初に更新してから2バイト目を更新します。

この2つのプログラムを同時に実行するとどうなるでしょうか。プログラムAは2バイト目をロックし、プログラムBは1バイト目をロックします。次に、プログラムAは1バイト目をロックしようとしますが、すでにプログラムBによってロックが設定されているため、プログラムAは待機することになります。一方、プログラムBは2バイト目をロックしようとしますが、すでにプログラムAによってロックが設定されているため、プログラムBも待機することになります。

このように、どちらのプログラムも処理を続行できない状態のことを**デッドロック**と呼びます。ほとんどの商用データベースはデッドロックを検出して回避しますが、UNIXカーネルはそういった処理を行いません。いずれかのプログラムを強制的に終了させるなど、外部からのなんらかの介入が必要になります。

プログラマは、こうした点について十分に注意する必要があります。ロック操作を行うために待機するプログラムが複数ある場合には、デッドロックが起きないかどうかを慎重に検討しなければなりません。ちなみに、上に例としてあげたプログラムAとBの場合には、ロックの対象となる1バイト目と2バイト目をどちらのプログラムでも同じ順序でロックするか、またはより広い範囲のリージョンをロックすれば、デッドロックを回避することができます。

メモ

本書では、並列プログラムに関するさまざまな問題については取り上げません。興味があれば、『Principles of Concurrent Programming』(M. Ben Ari著、Prentice Hall刊)などの専門書を参照してください。

7.3 データベース

ファイルを使ってデータを格納する方法についてはすでに取り上げましたが、データベースが提供する機能を使うと、プログラミング上のさまざまな課題をより適切に解決できることがあります。データをファイルに格納する場合と比べて、データベースを使うことには2つ利点があります。ひとつは、サイズの異なるデータレコードを格納できる点です。フラットな構造化されていないファイルの場合、この機能の実装はいささか困難です。もうひとつは、データベースを利用することでインデックスを使ったデータの格納や取得が可能になる点です。インデックスが単純なレコード番号であれば、フラットファイルでも同等の機能を簡単に実装できますが、データベースの場合には、インデックスがレコード番号である必要はありません。任意の文字列をインデックスに使用できる点は、データベースの非常に大きな利点です。

7.3.1 dbmデータベース

X/Openに準拠したバージョンのUNIXには、データベースが付属しています。このデータベースは、残念ながらANSI SQL準拠のデータベースではありませんが、データの格納や取得に使用できる有益なルーチンを提供しています。`dbm`データベースは、比較的静的なデータにインデックスを付けて格納するときに役立ちます。データベースの理論家は、`dbm`はデータベースではなく、単にインデックス付きファイル格納システムにすぎないと主張するかもしれません。しかし、X/Open仕様では`dbm`をデータベースとして扱っており、本書でもその見解を踏襲してデータベースと呼ぶことにします。Linuxの場合、ディストリビューションによっては、GNU `gdbm`が代わりに使われていることがあります。

`dbm`データベースでは、可変サイズのデータ構造をインデックスを使って格納することができ、インデックスを使ったり、単純にデータベースを走査したりして、格納したデータ構造を取得することができます。`dbm`データベースでは、エントリの作成にはやや時間がかかりますが、エントリの取得は高速に行うことができます。このため、`dbm`データベースは、頻繁にアクセスするものの、ほとんど更新を行わないデータに対して使った場合に大きな威力を発揮します。

7.3.2 dbmのしくみ

前の章で取り上げた`curses`同様、`dbm`の機能を使う場合にも適切なヘッダーファイルをインクルードし、ライブラリを指定する必要があります。ライブラリの名前は`dbm`なので、コンパイル時のコマンドラインにはオプション`-ldbm`(GNU `gdbm`を使っている場合には`-lgdbm`)を指定します。ヘッダーファイルは`ndbm.h`です。`dbm`ライブラリには先行バージョンがいくつも存在するので、古いバージョンのものと区別し、新しい`dbm`のインクルードファイルであることを示

すためにプレフィックスnが付いています。

dbmの関数について解説する前に、dbmデータベースがなにを行うものなのか簡単に説明しておきましょう。この点をおさえておけば、dbmの関数の使い方をスムーズに理解することができます。

dbmデータベースでは、データベースに格納するひとかたまりのデータ（データブロック）が基本要素になりますが、このデータを取得するためのキーとなるもうひとつのデータブロックを組み合わせて使います。dbmデータベースでは、データベースに格納される各データブロックに対して一意のキーが必要です。キーやデータに関する制限はなく、また、大きすぎるデータやキーを使った場合のエラーも定義されていません。キーの値は、データベースに格納されているデータのインデックスとして機能します。

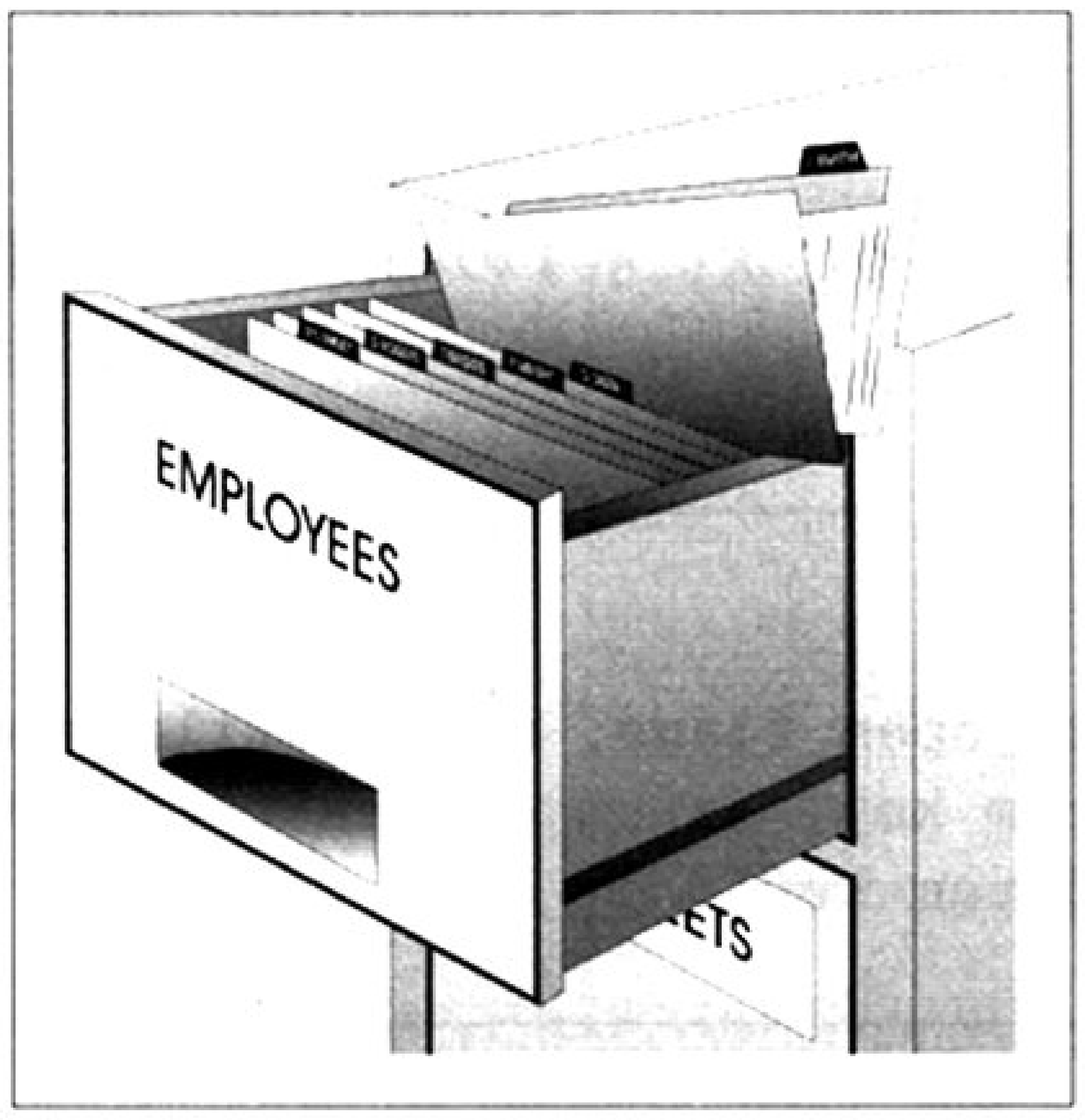


図7.2　dbmデータベースの概念

インクルードファイルndbm.hでは、データブロックをデータとして操作するためにdatumという新しい型が定義されています。この型の具体的な内容は処理系依存ですが、少なくとも次の2つのメンバが含まれています。

```
void *dptr;
size_t dsize
```

datumはtypedefによって定義されています。ndbm.hでは、このほかにDBMという型も定義されています。DBMはデータベースにアクセスするときに使われる構造体です。イメージとしては、ファイルにアクセスするときに使うFILE構造体を思い浮かべるとよいでしょう。DBM型の内部構造は処理系依存なので、DBM構造体の内部にアクセスしてはいけません。

dbmライブラリを使ってデータブロックを参照するには、datumを宣言し、dptrにデータの開始位置を示すポインタを、またdsizeにデータのサイズをセットします。格納するデータとこのデータにアクセスするときに使うインデックスとは、どちらも常にdatum型を通じて参照します。

すでに述べたように、DBM型はFILE型とよく似ています。dbmデータベースをオープンする

と、物理的なファイルが2つ作成されます。ひとつは拡張子が.pagのファイル、もうひとつは拡張子が.dirのファイルです。データベースをオープンしたときに返されるDBMポインタは1つで、このポインタを使って、2つのファイルをペアとして扱ってアクセスします。これら2つのファイルはdbmのルーチンを介してアクセスすることになっており、直接読み取りや書き込みを行ってはいけません。

すでにSQLデータベースについていくらか知識のある読者のために書き添えておくと、dbmデータベースにはテーブルやカラムといった構造は存在しません。dbmでは、格納する各データが固定サイズである必要はなく、データ自身が内部構造を持つ必要もないからです。dbmライブラリが対象としているのは、構造化されていないバイナリデータのかたまりです。

7.3.3 dbmのアクセス関数

さて、実際に関数について詳しく見ていきましょう。次に示すのは、主なdbm関数のプロトタイプです。

```
#include <ndbm.h>

DBM *dbm_open(const char *filename, int file_open_flags, mode_t file_mode);
int dbm_store(DBM *database_descriptor, datum key, datum content, int store_mode);
datum dbm_fetch(DBM *database_descriptor, datum key);
void dbm_close(DBM *database_descriptor);
```

◆ dbm_open

この関数は、既存のデータベースをオープンしたり、データベースを新しく作成したりするときに使います。引数filenameには、.dirや.pagの拡張子を付けずにベースファイル名を指定します。ベースファイル名の長さについては、4文字分の拡張子を付けてもファイル名の長さに関するシステムの制限を超えることがないように注意する必要があります。

残りの2つのパラメータは、第3章で説明したopen関数の第2パラメータおよび第3パラメータと同じで、パラメータに指定する定数も同じものを使用できます。2番目の引数は、読み取り、書き込み、読み書きのうち、どのモードでデータベースをオープンするかを指定します。新しいデータベースを作成するには、O_CREATとの論理和をフラグに指定してファイルを作成できるようにします。3番目の引数には、作成されるファイルの初期パーミッションを指定します。

dbm_openは、成功するとDBM型へのポインタを返します。以後は、このポインタを使ってデータベースにアクセスします。dbm_openは、失敗すると(DBM *)0を返します。

◆ dbm_store

dbm_store関数は、データベースにデータを入力するときに使います。すでに説明したように、すべてのデータは一意のインデックスを付けて格納しなければなりません。格納するデータと、データを参照するのに使うインデックスを定義するには、インデックス用と実際のデータ用の2つのdatum型を用意します。最後のパラメータstore_modeには、データを格納する際にキーがすでに存在していた場合の動作を指定します。DBM_INSERTを指定した場合、操作は失敗し、dbm_storeは1を返します。DBM_REPLACEを指定した場合、既存のデータは新しいデータで上書きされ、dbm_storeは0を返します。ほかのエラーが発生した場合、dbm_storeは負の値を返します。

◆ dbm_fetch

dbm_fetch関数は、データベースからデータを取得するときに使います。引数には、dbm_openから返されたDBMへのポインタと、キーを表すdatum型とを指定します。戻り値はdatum型です。指定したキーに関連付けられたデータがデータベース内に見つかった場合には、返されたdatum構造体のdptrとdsizeの値を使って、該当するデータを参照することができます。キーが見つからなかった場合、dptrにはNULLが設定されます。

dbm_fetchが返すのは、データへのポインタを含むdatumだけであり、実際のデータはdbmライブラリ内のローカルな記憶領域に保持されている可能性があることに注意する必要があります。したがって、引き続きdbmの関数を呼び出す場合には、その前にプログラムで用意した変数にデータの値をコピーしておく必要があります。

◆ dbm_close

この関数は、dbm_openによってオープンされたデータベースをクローズします。指定するパラメータは、dbm_openの呼び出しで返されたDBMへのポインタでなければなりません。

主な関数についてひととおり説明が済んだので、さっそくdbmを使ったプログラムを作成してみましょう。名前はdbm1.cとします。プログラムでは、test_dataという名前の構造体を使います。

例題 簡単なdbmデータベース

1 ヘッダーファイルをインクルードし、ファイル名を定義したあと、test_data構造体を宣言します。

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <ndbm.h>
```

```
#include <string.h>

#define TEST_DB_FILE "/tmp/dbm1_test"
#define ITEMS_USED 3

struct test_data {
    char misc_chars[15];
    int  any_integer;
    char more_chars[21];
};

int main()
{
```

2 main関数では、items_to_storeとitems_received構造体、キーとして使う文字列、およびdatum型の変数を用意します。

```
    struct test_data items_to_store[ITEMS_USED];
    struct test_data item_retrieved;

    char key_to_use[20];
    int i, result;

    datum key_datum;
    datum data_datum;

    DBM *dbm_ptr;
```

3 テストデータベースを読み書き用にオープンします。データベースが存在していない場合には作成します。

```
    dbm_ptr = dbm_open(TEST_DB_FILE, O_RDWR | O_CREAT, 0666);
    if (!dbm_ptr) {
        fprintf(stderr, "Failed to open database\n");
        exit(EXIT_FAILURE);
    }
```

4 items_to_store構造体にデータベースを設定します。

```
    memset(items_to_store, '\0', sizeof(items_to_store));
    strcpy(items_to_store[0].misc_chars, "First!");
    items_to_store[0].any_integer = 47;
    strcpy(items_to_store[0].more_chars, "foo");

    strcpy(items_to_store[1].misc_chars, "bar");
    items_to_store[1].any_integer = 13;
    strcpy(items_to_store[1].more_chars, "unlucky?");

    strcpy(items_to_store[2].misc_chars, "Third");
```

```
    items_to_store[2].any_integer = 3;
    strcpy(items_to_store[2].more_chars, "baz");
```

5 各エントリに対して、あとでデータを参照するときに使うキーを作成します。ここでは、文字列の先頭の文字と整数を組み合わせてキーにします。このキーはkey_datumで参照します。data_datumは、items_to_storeエントリを参照するのに使います。データをデータベースに格納します。

```
    for (i = 0; i < ITEMS_USED; i++) {
        sprintf(key_to_use, "%c%c%d",
                items_to_store[i].misc_chars[0],
                items_to_store[i].more_chars[0],
                items_to_store[i].any_integer);

        key_datum.dptr = (void *)key_to_use;
        key_datum.dsize = strlen(key_to_use);
        data_datum.dptr = (void *)&items_to_store[i];
        data_datum.dsize = sizeof(struct test_data);

        result = dbm_store(dbm_ptr, key_datum, data_datum, DBM_REPLACE);
        if (result != 0) {
            fprintf(stderr, "dbm_store failed on key %s\n", key_to_use);
            exit(2);
        }
    }
```

6 新しく格納したデータを取得できるかどうか調べ、最後にデータベースをクローズします。

```
    sprintf(key_to_use, "bu%d", 13
    key_datum.dptr = key_to_use;
    key_datum.dsize = strlen(key_to_use);

    data_datum = dbm_fetch(dbm_ptr, key_datum);
    if (data_datum.dptr) {
        printf("Data retrieved\n");
        memcpy(&item_retrieved, data_datum.dptr, data_datum.dsize);
        printf("Retrieved item - %s %d %s\n",
               item_retrieved.misc_chars,
               item_retrieved.any_integer,
               item_retrieved.more_chars);
    }
    else {
        printf("No data found for key %s\n", key_to_use);
    }
    dbm_close(dbm_ptr);
    exit(EXIT_SUCCESS);
}
```

プログラムをコンパイルして実行すると、次のように出力されます。

```
$ gcc -o dbm1 dbm1.c -ldbm
$ dbm1
Data retrieved
Retrieved item - bar 13 unlucky?
```

解説

最初にデータベースをオープンし、データベースが存在していない場合には作成します。次に、**items_to_store**の3つの要素に値を設定し、これらをテストデータとして使います。各要素に対してキーを作成しています。ここでは、ごく簡単に2つの文字列の先頭の文字と整数を組み合わせてキーにしています。

datum構造体を2つ用意します。ひとつはキー用、もうひとつはデータベースに格納するデータ用です。3つの要素をデータベースに格納したあと、新しいキーを作成し、このキーを指す**datum**構造体をセットアップします。次に、このキーを使ってデータベースからデータを取得します。データの取得が成功したかどうかを調べるために、戻り値の**datum**構造体の**dptr**が**NULL**ではないことを確かめます。**NULL**でなければ、取得したデータ（**dbm**ライブラリで内部的に格納している可能性があります）を、プログラムで用意した構造体にコピーします。このとき、**dbm_fetch**が返したサイズを指定します。最後に、取得したデータを表示します。

7.3.4 その他のdbm関数

dbmで使うその他の関数には次のようなものがあります。

```
int dbm_delete(DBM *database_descriptor, datum key);
int dbm_error(DBM *database_descriptor);
int dbm_clearerr(DBM *database_descriptor);
datum dbm_firstkey(DBM *database_descriptor);
datum dbm_nextkey(DBM *database_descriptor);
```

◆ dbm_delete

dbm_delete関数は、データベースからエントリを削除します。**dbm_fetch**同様、キーとなる**datum**を引数に指定します。成功すると0を返します。

◆ dbm_error

dbm_error関数は、データベースでエラーが発生しているかどうかを調べます。エラーが発生していない場合には0を返します。

◆ dbm_clearerr

dbm_clearerr関数は、データベース内のエラーフラグをクリアします。

◆ dbm_firstkeyとdbm_nextkey

通常、これら2つの関数は対で使用し、データベース内の全項目のすべてのキーを走査します。必要なループ構造は次のとおりです。

```
DBM *db_ptr;
datum key;

for(key = dbm_firstkey(db_ptr); key.dptr; key = dbm_nextkey(db_ptr));
```

次のサンプルプログラムdbm2.cでは、dbm1.cに修正を加え、上で説明したいくつかの新しい関数を使います。

例題 データの取得と削除

1 dbm1.cをコピーし、#define TEST_DB_FILEの行を編集します。

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <ndbm.h>
#include <string.h>

#define TEST_DB_FILE "/tmp/dbm2_test"
#define ITEMS_USED 3
```

2 データを取得する部分に変更を加えます。

```
        /* now try and delete some data */
    sprintf(key_to_use, "bu%d", 13);
    key_datum.dptr = key_to_use;
    key_datum.dsize = strlen(key_to_use);

    if (dbm_delete(dbm_ptr, key_datum) == 0) {
        printf("Data with key %s deleted\n", key_to_use);
    }
    else {
        printf("Nothing deleted for key %s\n", key_to_use);
    }

    for (key_datum = dbm_firstkey(dbm_ptr);
           key_datum.dptr;
           key_datum = dbm_nextkey(dbm_ptr)) {
        data_datum = dbm_fetch(dbm_ptr, key_datum);
```

```
        if (data_datum.dptr) {
            printf("Data retrieved\n");
            memcpy(&item_retrieved, data_datum.dptr, data_datum.dsize);
            printf("Retrieved item - %s %d %s\n",
                   item_retrieved.misc_chars,
                   item_retrieved.any_integer,
                   item_retrieved.more_chars);
        }
        else {
            printf("No data found for key %s\n", key_to_use);
        }
    }
```

プログラムを実行すると、次のように出力されます。

```
$ dbm2
Data with key bu13 deleted
Data retrieved
Retrieved item - Third 3 baz
Data retrieved
Retrieved item - First! 47 foo
```

解説

プログラムの前半部分は**dbm1.c**と同じで、いくつかのデータをデータベースに格納します。次に、2番目の要素に対応するキーを使ってこの要素をデータベースから削除します。

最後に、**dbm_firstkey**と**dbm_nextkey**を使ってデータベース内のすべてのキーにアクセスします。この操作では、単にデータベース内のすべてのエントリを走査するだけです。キーには特に順序はありません。

7.4 CDデータベースアプリケーション

データを管理するテクニックをいくつか学んだので、さっそくCDデータベースアプリケーションに応用することにしましょう。**dbm**データベースは、CD情報を格納するのに適しているように思われます。そこで、新しいCDデータベースアプリケーションでは**dbm**データベースを使うことにします。プログラムを大幅に書き直す必要がありますから、ついでに設計についても少し検討を加えておきましょう。

CSVファイルを使って情報を格納する方法は、シェルで簡単に実装できるものの、格納できるデータが制限されるという欠点があります。CDタイトルやトラック情報にはカンマを含むものも多くあるからです。**dbm**を使うと、この欠点を解消することができます。**dbm**を採用する価値は十分にありそうです。

タイトルとトラックの情報を区別し、それぞれ別のファイルに格納する方法は、アプローチとして正解だったようです。そこで、この方法はそのまま使うことにします。

これまでの実装では、データにアクセスする部分とユーザーインタフェースの部分を明確には区別していませんでした。これは、すべてを1つのファイルで実装していたことにも原因があります。そこで、データとデータにアクセスするルーチンとを宣言したヘッダーファイルを用意し、ユーザーインタフェースとデータ操作を行う部分とを独立したファイルとして作成することにします。

前の章では、ユーザーインタフェースの部分を実装するのにcursesを使いましたが、ここでは単純な行ベースのシステムに戻すことにします。これは、ユーザーインタフェースの部分をシンプルかつコンパクトにしておき、アプリケーションの本質的な部分に注意を集中できるようにするためです。

dbmを使ったコードでSQLが利用できるわけではありませんが、作成する新しいデータベースは、SQLの構文を使って的確に表現することができます。次に示すのはテーブルの定義です。SQLの知識がなくても、内容は容易に理解できるはずです。

```
CREATE TABLE cdc_entry (
    catalogue CHAR(30) PRIMARY KEY REFERENCES cdt_entry(catalogue),
    title     CHAR(70),
    type      CHAR(30),
    artist    CHAR(70)
);

CREATE TABLE cdt_entry (
    catalogue CHAR(30) REFERENCES cdc_entry(catalogue),
    track_no  INTEGER CHECK (track_no > 0),
    track_txt CHAR(70),
    PRIMARY KEY(catalogue, track_no)
);
```

上のSQL文では、フィールドの名前とサイズが定義されています。cdc_entryテーブルでは、各エントリに対して一意のカタログ列があります。また、cdt_entryテーブルでは、トラック番号が0より大きくなければならず、カタログ列とトラック番号列を組み合わせたものが一意であることがわかります。

7.4.1 dbmを使ったCDデータベースアプリケーション

では、実際にdbmデータベースを使ってアプリケーションを実装し直しましょう。アプリケーションを構成するファイルは、cd_data.h、app_ui.c、cd_access.cの3つです。

すでに触れたように、ユーザーインタフェースはコマンドラインベースに戻します。あとの章では、データベースインタフェースとユーザーインタフェースの一部を再利用し、クライアント

サーバー方式のアプリケーションの実装を試みます。もちろん、ユーザーインタフェースにさらに工夫を加え、WWWブラウザを使ってネットワーク経由でアプリケーションにアクセスできるようにすることも可能です。ここでは、シンプルなコマンドラインインタフェースにしておくことで、アプリケーションの重要な部分に力を注ぐことにします。

ヘッダーファイルの`cd_data.h`と、`cd_access.c`ファイル内のいくつかの関数は、本書の以後の章でも繰り返し使われます。ここではまず、データの構造とデータにアクセスするためのルーチンとを定義したヘッダーファイルから取り上げます。

例題 cd_data.h

1 CDデータベースのデータ構造を定義します。具体的には、データベースを構成する2つのテーブルの構造とサイズを定義します。まず、各フィールドのサイズを定義し、カタログエントリ用とトラックエントリ用の2つの構造体を定義します。

```
/* The catalogue table */
#define CAT_CAT_LEN       30
#define CAT_TITLE_LEN     70
#define CAT_TYPE_LEN      30
#define CAT_ARTIST_LEN    70

typedef struct {
    char catalog[CAT_CAT_LEN + 1];
    char title[CAT_TITLE_LEN + 1];
    char type[CAT_TYPE_LEN + 1];
    char artist[CAT_ARTIST_LEN + 1];
} cdc_entry;

/* The tracks table, one entry per track */
#define TRACK_CAT_LEN     CAT_CAT_LEN
#define TRACK_TTEXT_LEN   70

typedef struct {
    char catalog[TRACK_CAT_LEN + 1];
    int  track_no;
    char track_txt[TRACK_TTEXT_LEN + 1];
} cdt_entry;
```

2 テーブルにアクセスするためのルーチンを定義します。`cdc_`で始まる関数はカタログエントリ用、`cdt_`で始まる関数はトラックエントリ用です。

注意

> いくつかの関数は、上で定義した構造体を返します。この場合、構造体の内容を空にすることでエラーを表現します。

```
/* Initialization and termination functions */
int database_initialize(const int new_database);
void database_close(void);

/* two for simple data retrieval */
cdc_entry get_cdc_entry(const char *cd_catalog_ptr);
cdt_entry get_cdt_entry(const char *cd_catalog_ptr, const int
track_no);

/* two for data addition */
int add_cdc_entry(const cdc_entry entry_to_add);
int add_cdt_entry(const cdt_entry entry_to_add);

/* two for data deletion */
int del_cdc_entry(const char *cd_catalog_ptr);
int del_cdt_entry(const char *cd_catalog_ptr, const int track_no);

/* one search function */
cdc_entry search_cdc_entry(const char *cd_catalog_ptr, int *first_call_ptr);
```


例題 app_ui.c

1 app_ui.cでは、データベース関数にアクセスするためのユーザーインタフェースを実装します。データベース関数自体は別のファイルで実装します。まず、必要なヘッダーファイルをインクルードします。

```
#define _XOPEN_SOURCE

#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

#include "cd_data.h"

#define TMP_STRING_LEN 125 /* this number must be larger than the biggest
                              single string in any database structure */
```

2 メニューオプションは、typedefを使って定義します。#defineではなくtypedefを使うことで、メニューオプション変数の型をコンパイラがチェックできるようになります。

```
typedef enum {
    mo_invalid,
    mo_add_cat,
    mo_add_tracks,
    mo_del_cat,
    mo_find_cat,
```

```
    mo_list_cat_tracks,
    mo_del_tracks,
    mo_count_entries,
    mo_exit
} menu_options;
```

3 ローカル関数のプロトタイプを宣言します。実際にデータベースにアクセスする関数のプロトタイプは`cd_data.h`に含まれています。

```
static int command_mode(int argc, char *argv[]);
static void announce(void);
static menu_options show_menu(const cdc_entry *current_cdc);
static int get_confirm(const char *question);
static int enter_new_cat_entry(cdc_entry *entry_to_update);
static void enter_new_track_entries(const cdc_entry *entry_to_add_to);
static void del_cat_entry(const cdc_entry *entry_to_delete);
static void del_track_entries(const cdc_entry *entry_to_delete);
static cdc_entry find_cat(void);
static void list_tracks(const cdc_entry *entry_to_use);
static void count_all_entries(void);
static void display_cdc(const cdc_entry *cdc_to_show);
static void display_cdt(const cdt_entry *cdt_to_show);
static void strip_return(char *string_to_strip);
```

4 `main`関数です。まず、現在選択されているCDカタログエントリを記録しておくのに使う`current_cdc_entry`を初期化します。また、コマンドラインを解析し、プログラムの内容を簡単に表示してデータベースを初期化します。

```
void main(int argc, char *argv[])
{
    menu_options current_option;
    cdc_entry current_cdc_entry;
    int command_result;

    memset(&current_cdc_entry, '\0', sizeof(current_cdc_entry));

    if (argc > 1) {
        command_result = command_mode(argc, argv);
        exit(command_result);
    }

    announce();

    if (!database_initialize(0)) {
        fprintf(stderr, "Sorry, unable to initialize database\n");
        fprintf(stderr, "To create a new database use %s -i\n",
argv[0]);
        exit(EXIT_FAILURE);
    }
```

5 ユーザーからの入力を処理します。ループを用意してメニューの選択を求め、選択されたメニューを処理します。ユーザーが終了を選択すると、ループを抜けてプログラムを終了します。show_menu関数にはcurrent_cdc_entry構造体を渡し、カタログエントリが現在選択されている場合には、表示するメニューを変更できるようにしています。

```
while(current_option != mo_exit) {
    current_option = show_menu(&current_cdc_entry);

    switch(current_option) {
        case mo_add_cat:
            if (enter_new_cat_entry(&current_cdc_entry)) {
                if (!add_cdc_entry(current_cdc_entry)) {
                    fprintf(stderr, "Failed to add new entry\n");
                    memset(&current_cdc_entry, '\0',
                           sizeof(current_cdc_entry));
                }
            }
            break;
        case mo_add_tracks:
            enter_new_track_entries(&current_cdc_entry);
            break;
        case mo_del_cat:
            del_cat_entry(&current_cdc_entry);
            break;
        case mo_find_cat:
            current_cdc_entry = find_cat();
            break;
        case mo_list_cat_tracks:
            list_tracks(&current_cdc_entry);
            break;
        case mo_del_tracks:
            del_track_entries(&current_cdc_entry);
            break;
        case mo_count_entries:
            count_all_entries();
            break;
        case mo_exit:
            break;
        case mo_invalid:
            break;
        default:
            break;
    } /* switch */
} /* while */
```

6 ループを抜けたら、データベースをクローズしてプログラムを終了します。announce関数は、プログラムの内容を簡単に表示します。

```
database_close();
exit(EXIT_SUCCESS);
```

```
} /* main */

static void announce(void)
{
    printf("\n\nWelcome to the demonstration CD catalog database \
            program\n");
}
```

7 show_menu関数です。この関数では、カタログ名の最初の文字を使って、カタログエントリが現在選択されているかどうかをチェックします。カタログエントリが選択されている場合には、選択可能なオプションが増えます。

> メモ
> これまでに作成したCDデータベースアプリケーションでは、メニュー項目を選択するのに項目の先頭の文字を使っていましたが、ここでは数字を使います。

```
static menu_options show_menu(const cdc_entry *cdc_selected)
{
    char tmp_str[TMP_STRING_LEN + 1];
    menu_options option_chosen = mo_invalid;

    while (option_chosen == mo_invalid) {
        if (cdc_selected->catalog[0]) {
            printf("\n\nCurrent entry: ");
            printf("%s, %s, %s, %s\n", cdc_selected->catalog,
                   cdc_selected->title,
                   cdc_selected->type,
                   cdc_selected->artist);

            printf("\n");
            printf("1 - add new CD\n");
            printf("2 - search for a CD\n");
            printf("3 - count the CDs and tracks in the database\n");
            printf("4 - re-enter tracks for current CD\n");
            printf("5 - delete this CD, and all its tracks\n");
            printf("6 - list tracks for this CD\n");
            printf("q - quit\n");
            printf("\nOption: ");
            fgets(tmp_str, TMP_STRING_LEN, stdin);
            switch(tmp_str[0]) {
                case '1': option_chosen = mo_add_cat; break;
                case '2': option_chosen = mo_find_cat; break;
                case '3': option_chosen = mo_count_entries; break;
                case '4': option_chosen = mo_add_tracks; break;
                case '5': option_chosen = mo_del_cat; break;
                case '6': option_chosen = mo_list_cat_tracks; break;
                case 'q': option_chosen = mo_exit; break;
            }
        }
```

```
        else {
            printf("\n\n");
            printf("1 - add new CD\n");
            printf("2 - search for a CD\n");
            printf("3 - count the CDs and tracks in the database\n");
            printf("q - quit\n");
            printf("\nOption: ");
            fgets(tmp_str, TMP_STRING_LEN, stdin);
            switch(tmp_str[0]) {
                case '1': option_chosen = mo_add_cat; break;
                case '2': option_chosen = mo_find_cat; break;
                case '3': option_chosen = mo_count_entries; break;
                case 'q': option_chosen = mo_exit; break;
            }
        }
    } /* while */
    return(option_chosen);
}
```


8 プログラムの中でユーザーからの要求内容を確認したいことがあります。このような場合に使用できる関数として`get_confirm`を用意します。

```
static int get_confirm(const char *question)
{
    char tmp_str[TMP_STRING_LEN + 1];

    printf("%s", question);
    fgets(tmp_str, TMP_STRING_LEN, stdin);
    if (tmp_str[0] == 'Y' || tmp_str[0] == 'y') {
        return(1);
    }
    return(0);
}
```

9 `enter_new_cat_entry`関数は、ユーザーが新しいカタログエントリを入力するときに使います。`fgets`が返すラインフィードはデータに含めず、捨てます。

> **注意**
> `gets`関数を使うとバッファのオーバフローをチェックできないので、`fgets`関数を使っていることに注意してください。一般に、`gets`関数の使用は避けるべきです。

```
static int enter_new_cat_entry(cdc_entry *entry_to_update)
{
    cdc_entry new_entry;
    char tmp_str[TMP_STRING_LEN + 1];

    memset(&new_entry, '\0', sizeof(new_entry));
```

```
    printf("Enter catalog entry: ");
    (void)fgets(tmp_str, TMP_STRING_LEN, stdin);
    strip_return(tmp_str);
    strncpy(new_entry.catalog, tmp_str, CAT_CAT_LEN - 1);

    printf("Enter title: ");
    (void)fgets(tmp_str, TMP_STRING_LEN, stdin);
    strip_return(tmp_str);
    strncpy(new_entry.title, tmp_str, CAT_TITLE_LEN - 1);

    printf("Enter type: ");
    (void)fgets(tmp_str, TMP_STRING_LEN, stdin);
    strip_return(tmp_str);
    strncpy(new_entry.type, tmp_str, CAT_TYPE_LEN - 1);

    printf("Enter artist: ");
    (void)fgets(tmp_str, TMP_STRING_LEN, stdin);
    strip_return(tmp_str);
    strncpy(new_entry.artist, tmp_str, CAT_ARTIST_LEN - 1);

    printf("\nNew catalog entry entry is :-\n");
    display_cdc(&new_entry);
    if (get_confirm("Add this entry ?")) {
        memcpy(entry_to_update, &new_entry, sizeof(new_entry));
        return(1);
    }
    return(0);
}
```

10 トラック情報を入力するためのenter_new_track_entries関数です。既存のトラックエントリを残す必要があるので、カタログエントリ関数よりやや複雑になっています。

```
static void enter_new_track_entries(const cdc_entry *entry_to_add_to)
{
    cdt_entry new_track, existing_track;
    char tmp_str[TMP_STRING_LEN + 1];
    int track_no = 1;

    if (entry_to_add_to->catalog[0] == '\0') return;
    printf("\nUpdating tracks for %s\n", entry_to_add_to->catalog);
    printf("Press return to leave existing description unchanged,\n");
    printf(" a single d to delete this and remaining tracks,\n");
    printf(" or new track description\n");

    while(1) {
```

11 まず、現在のトラック番号に対応するトラックがすでに存在するかどうかをチェックする必要があります。その結果に応じてプロンプトを変えます。

```
memset(&new_track, '\0', sizeof(new_track));
existing_track = get_cdt_entry(entry_to_add_to->catalog,
                                track_no);
if (existing_track.catalog[0]) {
    printf("\tTrack %d: %s\n", track_no,
                existing_track.track_txt);
    printf("\tNew text: ");
}
else {
    printf("\tTrack %d description: ", track_no);
}
fgets(tmp_str, TMP_STRING_LEN, stdin);
strip_return(tmp_str);
```

12 トラックに既存のエントリが存在せず、ユーザーがエントリを追加しなかった場合には、これ以上追加するトラックはないものと判断します。

```
if (strlen(tmp_str) == 0) {
    if (existing_track.catalog[0] == '\0') {
            /* no existing entry, so finished adding */
        break;
    }
    else {
        /* leave existing entry, jump to next track */
        track_no++;
        continue;
    }
}
```

13 ユーザーが1つだけdを入力した場合には、現在のトラックとそれ以降のトラックを削除します。del_cdt_entry関数は、削除するトラックが見つからないと偽を返します。

```
if ((strlen(tmp_str) == 1) && tmp_str[0] == 'd') {
        /* delete this and remaining tracks */
    while (del_cdt_entry(entry_to_add_to->catalog, track_no)  {
        track_no++;
    }
    break;
}
```

14 次の部分では、新しいトラックの追加や既存のトラックの更新を行います。cdt_entry構造体new_trackを作成し、データベース関数add_cdt_entryを呼び出してデータベースへの追加を行います。

```
strncpy(new_track.track_txt, tmp_str, TRACK_TTEXT_LEN - 1);
strcpy(new_track.catalogue, entry_to_add_to->catalog);
new_track.track_no = track_no;
if (!add_cdt_entry(new_track)) {
```

```
                fprintf(stderr, "Failed to add new track\n");
                break;
            }
            track_no++;
        } /* while */
}
```

15 del_cat_entry関数はカタログエントリを削除します。このとき、カタログエントリとともにトラックエントリも削除します。

```
static void del_cat_entry(const cdc_entry *entry_to_delete)
{
    int track_no = 1;
    int delete_ok;

    display_cdc(entry_to_delete);
    if (get_confirm("Delete this entry and all it's tracks? ")) {
        do {
            delete_ok = del_cdt_entry(entry_to_delete->catalog,
                                      track_no);
            track_no++;
        } while(delete_ok);

        if (!del_cdc_entry(entry_to_delete->catalog)) {
            fprintf(stderr, "Failed to delete entry\n");
        }
    }
}
```

16 次の関数は、1つのカタログに含まれるすべてのトラックを削除するためのユーティリティ関数です。

```
static void del_track_entries(const cdc_entry *entry_to_delete)
{
    int track_no = 1;
    int delete_ok;

    display_cdc(entry_to_delete);
    if (get_confirm("Delete tracks for this entry? ")) {
        do {
            delete_ok = del_cdt_entry(entry_to_delete->catalog, track_no);
            track_no++;
        } while(delete_ok);
    }
}
```

17 次の関数は、非常に簡単なカタログ検索機能を提供する関数です。ユーザーから入力文字列を受け取り、その文字列を含むカタログエントリを探します。一致するエントリが複数存在する可能性があるので、一致するたびに内容を表示し、ユーザーに確認を求めます。

```
static cdc_entry find_cat(void)
{
    cdc_entry item_found;
    char tmp_str[TMP_STRING_LEN + 1];
    int first_call = 1;
    int any_entry_found = 0;
    int string_ok;
    int entry_selected = 0;

    do {
        string_ok = 1;
        printf("Enter string to search for in catalog entry: ");
        fgets(tmp_str, TMP_STRING_LEN, stdin);
        strip_return(tmp_str);
        if (strlen(tmp_str) > CAT_CAT_LEN) {
            fprintf(stderr, "Sorry, string too long, maximum %d \
                            characters\n", CAT_CAT_LEN);
            string_ok = 0;
        }
    } while (!string_ok);

    while (!entry_selected) {
        item_found = search_cdc_entry(tmp_str, &first_call);
        if (item_found.catalog[0] != '\0') {
            any_entry_found = 1;
            printf("\n");
            display_cdc(&item_found);
            if (get_confirm("This entry? ")) {
                entry_selected = 1;
            }
        }
        else {
            if (any_entry_found) printf("Sorry, no more matches found\n");
            else printf("Sorry, nothing found\n");
            break;
        }
    }
    return(item_found);
}
```

18 list_tracks関数は、指定されたカタログエントリのすべてのトラックを表示するユーティリティ関数です。

```
static void list_tracks(const cdc_entry *entry_to_use)
{
    int track_no = 1;
    cdt_entry entry_found;

    display_cdc(entry_to_use);
    printf("\nTracks\n");
    do {
            entry_found = get_cdt_entry(entry_to_use->catalog,
```

```
                                        track_no);
            if (entry_found.catalog[0]) {
                display_cdt(&entry_found);
                track_no++;
            }
    } while(entry_found.catalog[0]);
    (void)get_confirm("Press return");
} /* list_tracks */
```

19 count_all_entries関数は、すべてのトラックをカウントします。

```
static void count_all_entries(void)
{
    int cd_entries_found = 0;
    int track_entries_found = 0;
    cdc_entry cdc_found;
    cdt_entry cdt_found;
    int track_no = 1;
    int first_time = 1;
    char *search_string = "";

    do {
        cdc_found = search_cdc_entry(search_string, &first_time);
        if (cdc_found.catalog[0]) {
            cd_entries_found++;
            track_no = 1;
            do {
                cdt_found = get_cdt_entry(cdc_found.catalogue, track_no);
                if (cdt_found.catalog[0]) {
                    track_entries_found++;
                    track_no++;
                }
            } while (cdt_found.catalog[0]);
        }
    } while (cdc_found.catalog[0]);

    printf("Found %d CDs, with a total of %d tracks\n", cd_entries_found,
                track_entries_found);
    (void)get_confirm("Press return");
}
```

20 display_cdc関数は、カタログエントリを表示するユーティリティ関数です。

```
static void display_cdc(const cdc_entry *cdc_to_show)
{
    printf("Catalog: %s\n", cdc_to_show->catalog);
    printf("\ttitle: %s\n", cdc_to_show->title);
    printf("\ttype: %s\n", cdc_to_show->type);
    printf("\tartist: %s\n", cdc_to_show->artist);
}
```

同様に、display_cdt関数は、トラックエントリを1つ表示するユーティリティ関数です。

```
static void display_cdt(const cdt_entry *cdt_to_show)
{
    printf("%d: %s\n", cdt_to_show->track_no, cdt_to_show->track_txt);
}
```

21 ユーティリティ関数strip_returnは、文字列の最後のラインフィードを削除します。UNIXでは、ラインフィード単独で行の終わりを表します。

```
static void strip_return(char *string_to_strip)
{
    int len;

    len = strlen(string_to_strip);
    if (string_to_strip[len - 1] == '\n') string_to_strip[len - 1] = '\0';
}
```

22 command_mode関数は、コマンドライン引数を解析します。getopt関数を使うことで、UNIXの標準的なオプション指定方法を利用できるようにしています。

```
static int command_mode(int argc, char *argv[])
{
    int c;
    int result = EXIT_SUCCESS;
    char *prog_name = argv[0];

    /* these externals used by getopt */
    extern char *optarg;
    extern optind, opterr, optopt;

    while ((c = getopt(argc, argv, ":i")) != -1) {
        switch(c) {
            case 'i':
                if (!database_initialize(1)) {
                    result = EXIT_FAILURE;
                    fprintf(stderr, "Failed to initialize database\n");
                }
                break;
            case ':':
            case '?':
            default:
                fprintf(stderr, "Usage: %s [-i]\n", prog_name);
                result = EXIT_FAILURE;
                break;
        } /* switch */
    } /* while */
    return(result);
}
```

例題 cd_access.c

1 cd_access.cでは、dbmデータベースにアクセスするための関数を実装します。まず、ヘッダーファイルをインクルードし、データを格納するのに使うファイルの名前を定義します。

```
#define _XOPEN_SOURCE

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <ndbm.h>

#include "cd_data.h"

#define CDC_FILE_BASE "cdc_data"
#define CDT_FILE_BASE "cdt_data"
#define CDC_FILE_DIR  "cdc_data.dir"
#define CDC_FILE_PAG  "cdc_data.pag"
#define CDT_FILE_DIR "cdt_data.dir"
#define CDT_FILE_PAG "cdt_data.pag"
```

2 現在のデータベースへのポインタを格納するstatic変数を用意します。

```
static DBM *cdc_dbm_ptr = NULL;
static DBM *cdt_dbm_ptr = NULL;
```

3 database_initialize関数は、デフォルトでは既存のデータベースをオープンしますが、パラメータnew_databaseに真（0以外の値）を指定すれば、強制的に新しい（空の）データベースを作成することができます。データベースが正常に初期化されると、2つのデータベースポインタも初期化されます。これらのポインタを見ることで、データベースがオープンされているかどうかがわかります。

```
int database_initialize(const int new_database)
{
    int open_mode = O_RDWR;

    /* If any existing database is open then close it */
    if (cdc_dbm_ptr) dbm_close(cdc_dbm_ptr);
    if (cdt_dbm_ptr) dbm_close(cdt_dbm_ptr);

    if (new_database) {
        /* delete the old files */
        (void) unlink(CDC_FILE_PAG);
        (void) unlink(CDC_FILE_DIR);
        (void) unlink(CDT_FILE_PAG);
        (void) unlink(CDT_FILE_DIR);
```

```
        open_mode = O_CREAT | O_RDWR;
    }

    /* Open some new files, creating them if required */
    cdc_dbm_ptr = dbm_open(CDC_FILE_BASE, open_mode, 0644);
    cdt_dbm_ptr = dbm_open(CDT_FILE_BASE, open_mode, 0644);
    if (!cdc_dbm_ptr || !cdt_dbm_ptr) {
        fprintf(stderr, "Unable to create database\n");
        cdc_dbm_ptr = cdt_dbm_ptr = NULL;
        return (0);
    }
    return (1);
}
```

4 database_close関数は、データベースがオープンされていればクローズし、2つのデータベースポインタにNULLをセットします。これで、現在オープンされているデータベースがないことがわかります。

```
void database_close(void)
{
    if (cdc_dbm_ptr) dbm_close(cdc_dbm_ptr);
    if (cdt_dbm_ptr) dbm_close(cdt_dbm_ptr);

    cdc_dbm_ptr = cdt_dbm_ptr = NULL;
}
```

5 次は、カタログエントリを1つ取得する関数です。引数には、カタログのテキスト文字列へのポインタを指定します。エントリが見つからなかった場合、返されるデータのカタログフィールドは空になります。

```
cdc_entry get_cdc_entry(const char *cd_catalog_ptr)
{
    cdc_entry entry_to_return;
    char entry_to_find[CAT_CAT_LEN + 1];
    datum local_data_datum;
    datum local_key_datum;

    memset(&entry_to_return, '\0', sizeof(entry_to_return));
```

6 データベースがオープンされているかどうか、渡されたパラメータが適切かどうか（検索キーが有効な文字列かどうか）をチェックします。

```
    if (!cdc_dbm_ptr || !cdt_dbm_ptr) return (entry_to_return);
    if (!cd_catalog_ptr) return (entry_to_return);
    if (strlen(cd_catalog_ptr) >= CAT_CAT_LEN) return (entry_to_return);

    memset(&entry_to_find, '\0', sizeof(entry_to_find));
    strcpy(entry_to_find, cd_catalog_ptr);
```

7 dbm関数を呼び出すために必要なdatum構造体をセットアップし、dbm_fetchを使ってデータを取得します。データが見つからなかった場合には、初期化されたままの空のentry_to_return構造体を返します。

```
    local_key_datum.dptr = (void *) entry_to_find;
    local_key_datum.dsize = sizeof(entry_to_find);

    memset(&local_data_datum, '\0', sizeof(local_data_datum));
    local_data_datum = dbm_fetch(cdc_dbm_ptr, local_key_datum);
    if (local_data_datum.dptr) {
        memcpy(&entry_to_return, (char *)local_data_datum.dptr,
                   local_data_datum.dsize);
    }
    return (entry_to_return);
} /* get_cdc_entry */
```

8 get_cdc_entryと同じ要領で、トラックエントリを取得するget_cdt_entry関数を用意します。引数には、カタログ文字列へのポインタとトラック番号を指定します。

```
cdt_entry get_cdt_entry(const char *cd_catalog_ptr, const int track_no)
{
    cdt_entry entry_to_return;
    char entry_to_find[CAT_CAT_LEN + 10];
    datum local_data_datum;
    datum local_key_datum;

    memset(&entry_to_return, '\0', sizeof(entry_to_return));

    if (!cdc_dbm_ptr || !cdt_dbm_ptr) return (entry_to_return);
    if (!cd_catalog_ptr) return (entry_to_return);
    if (strlen(cd_catalog_ptr) >= CAT_CAT_LEN) return (entry_to_return);

    /* setup the search key, which is a composite key of catalog entry
       and track number */
    memset(&entry_to_find, '\0', sizeof(entry_to_find));
    sprintf(entry_to_find, "%s %d", cd_catalog_ptr, track_no);

    local_key_datum.dptr = (void *) entry_to_find;
    local_key_datum.dsize = sizeof(entry_to_find);

    memset(&local_data_datum, '\0', sizeof(local_data_datum));
    local_data_datum = dbm_fetch(cdt_dbm_ptr, local_key_datum);
    if (local_data_datum.dptr) {
        memcpy(&entry_to_return, (char *) local_data_datum.dptr,
                   local_data_datum.dsize);
    }
    return (entry_to_return);
}
```

9 add_cdc_entry関数は、新しくカタログエントリを追加します。

```
int add_cdc_entry(const cdc_entry entry_to_add)
{
    char key_to_add[CAT_CAT_LEN + 1];
    datum local_data_datum;
    datum local_key_datum;
    int result;

    /* check database initialized and parameters valid */
    if (!cdc_dbm_ptr || !cdt_dbm_ptr) return (0);
    if (strlen(entry_to_add.catalog) >= CAT_CAT_LEN) return (0);

    /* ensure the search key contains only the valid string and nulls */
    memset(&key_to_add, '\0', sizeof(key_to_add));
    strcpy(key_to_add, entry_to_add.catalog);

    local_key_datum.dptr = (void *) key_to_add;
    local_key_datum.dsize = sizeof(key_to_add);
    local_data_datum.dptr = (void *) &entry_to_add;
    local_data_datum.dsize = sizeof(entry_to_add);

    result = dbm_store(cdc_dbm_ptr, local_key_datum, local_data_datum,
                       DBM_REPLACE);

    /* dbm_store() uses 0 for success */
    if (result == 0) return (1);
    return (0);
}
```

10 add_cdt_entry関数は、新しくトラックエントリを追加します。アクセスキーには、カタログ文字列とトラック番号を組み合わせて使います。

```
int add_cdt_entry(const cdt_entry entry_to_add)
{
    char key_to_add[CAT_CAT_LEN + 10];
    datum local_data_datum;
    datum local_key_datum;
    int result;

    if (!cdc_dbm_ptr || !cdt_dbm_ptr) return (0);
    if (strlen(entry_to_add.catalog) >= CAT_CAT_LEN) return (0);

    memset(&key_to_add, '\0', sizeof(key_to_add));
    sprintf(key_to_add, "%s %d", entry_to_add.catalog,
            entry_to_add.track_no);

    local_key_datum.dptr = (void *) key_to_add;
    local_key_datum.dsize = sizeof(key_to_add);
    local_data_datum.dptr = (void *) &entry_to_add;
    local_data_datum.dsize = sizeof(entry_to_add);
```

```
    result = dbm_store(cdt_dbm_ptr, local_key_datum, local_data_datum,
                        DBM_REPLACE);

    /* dbm_store() uses 0 for success and -ve numbers for errors */
    if (result == 0)
        return (1);
    return (0);
}
```

11 del_cdc_entry関数は、カタログエントリを削除します。

```
int del_cdc_entry(const char *cd_catalog_ptr)
{
    char key_to_del[CAT_CAT_LEN + 1];
    datum local_key_datum;
    int result;

    if (!cdc_dbm_ptr || !cdt_dbm_ptr) return (0);
    if (strlen(cd_catalog_ptr) >= CAT_CAT_LEN) return (0);

    memset(&key_to_del, '\0', sizeof(key_to_del));
    strcpy(key_to_del, cd_catalog_ptr);

    local_key_datum.dptr = (void *) key_to_del;
    local_key_datum.dsize = sizeof(key_to_del);

    result = dbm_delete(cdc_dbm_ptr, local_key_datum);

    /* dbm_delete() uses 0 for success */
    if (result == 0) return (1);
    return (0);
}
```

12 del_cdt_entry関数は、トラックを削除します。トラックのキーとしてカタログエントリの文字列とトラック番号を組み合わせて使っているので、これらを引数に指定します。

```
int del_cdt_entry(const char *cd_catalog_ptr, const int track_no)
{
    char key_to_del[CAT_CAT_LEN + 10];
    datum local_key_datum;
    int result;

    if (!cdc_dbm_ptr || !cdt_dbm_ptr) return (0);
    if (strlen(cd_catalog_ptr) >= CAT_CAT_LEN) return (0);

    memset(&key_to_del, '\0', sizeof(key_to_del));
    sprintf(key_to_del, "%s %d", cd_catalog_ptr, track_no);

    local_key_datum.dptr = (void *) key_to_del;
    local_key_datum.dsize = sizeof(key_to_del);
```

```
    result = dbm_delete(cdt_dbm_ptr, local_key_datum);

    /* dbm_delete() uses 0 for success */
    if (result == 0) return (1);
    return (0);
}
```

13 最後は簡単な検索関数です。高機能ではありませんが、キーがわからない場合にdbmデータベースのエントリを走査するひとつの方法として用意しました。

エントリがいくつ存在するかを前もって知ることはできません。そこで、この関数では、呼び出されるたびにエントリを1つ返すことにします。エントリが見つからなかった場合には空のエントリを返します。データベース全体を走査するには、最初にこの関数を呼び出すときに、`*first_call_ptr`に1を設定します。これにより、呼び出された関数の側では、データベースの最初から検索を開始する必要があることがわかります。以後の呼び出しでは、`*first_call_ptr`が0になり、前回見つかったエントリの次のエントリが検索されることになります。

別のカタログエントリを使いたい場合など、検索をやり直したいときには、`*first_call_ptr`に真を設定して関数を呼び出します。これで検索処理が再初期化されます。

この関数は、何度か呼び出される間も内部状態に関する情報を保持しています。このようにすると、検索を続行する際のこまごまとした側面をクライアントから隠し、検索関数がどのように実装されているかを隠蔽することができます。

検索テキストがヌル文字を指している場合には、すべてのエントリがマッチするものとして処理されます。

```
cdc_entry search_cdc_entry(const char *cd_catalog_ptr, int *first_call_ptr)
{
    static int local_first_call = 1;
    cdc_entry entry_to_return;
    datum local_data_datum;
    static datum local_key_datum;    /* notice this must be static */

    memset(&entry_to_return, '\0', sizeof(entry_to_return));
```

14 まず、引数などが適切かどうかをチェックします。

```
    if (!cdc_dbm_ptr || !cdt_dbm_ptr) return (entry_to_return);
    if (!cd_catalog_ptr || !first_call_ptr) return (entry_to_return);
    if (strlen(cd_catalog_ptr) >= CAT_CAT_LEN) return (entry_to_return);

    /* protect against never passing *first_call_ptr true */
    if (local_first_call) {
        local_first_call = 0;
        *first_call_ptr = 1;
    }
```

15 *first_call_ptrに真が設定されてこの関数が呼び出された場合には、データベースの最初から検索を開始する必要があります。*first_call_ptrが真でなかった場合には、データベース内の次のキーを処理します。

```
    if (*first_call_ptr) {
        *first_call_ptr = 0;
        local_key_datum = dbm_firstkey(cdc_dbm_ptr);
    }
    else {
        local_key_datum = dbm_nextkey(cdc_dbm_ptr);
    }

    do {
        if (local_key_datum.dptr != NULL) {
            /* an entry was found */
            local_data_datum = dbm_fetch(cdc_dbm_ptr, local_key_datum);
            if (local_data_datum.dptr) {
                memcpy(&entry_to_return, (char *) local_data_datum.dptr,
local_data_datum.dsize);
```

16 検索機能自体は、検索文字列が現在のカタログエントリ内に存在するかどうかを調べるごく簡単なものです。

```
                /* check if search string occurs in the entry */
                if (!strstr(entry_to_return.catalog, cd_catalog_ptr))
                    {
                    memset(&entry_to_return, '\0',
                                sizeof(entry_to_return));
                    local_key_datum = dbm_nextkey(cdc_dbm_ptr);
                }
            }
        }
    } while (local_key_datum.dptr &&
            local_data_datum.dptr &&
            (entry_to_return.catalog[0] == '\0'));
    return (entry_to_return);
} /* search_cdc_entry */
```

さて、以上でプログラムの記述は終わりです。次に、makefileを作成します。次のような内容のファイルを作成し、Makefileという名前で保存します。makefileについては、次の章で詳しく説明します。

```
all:    application

app_ui.o: app_ui.c cd_data.h
        gcc -pedantic -Wall -ansi -g -c app_ui.c

cd_access.o: cd_access.c cd_data.h
```

```
	gcc -pedantic -Wall -ansi -g -c cd_access.c

application:	app_ui.o cd_access.o
	gcc -o application -pedantic -Wall -ansi -g app_ui.o cd_access.o -ldbm
```

CDデータベースアプリケーションをコンパイルするには、次のように入力します。

```
$ make -f Makefile
```

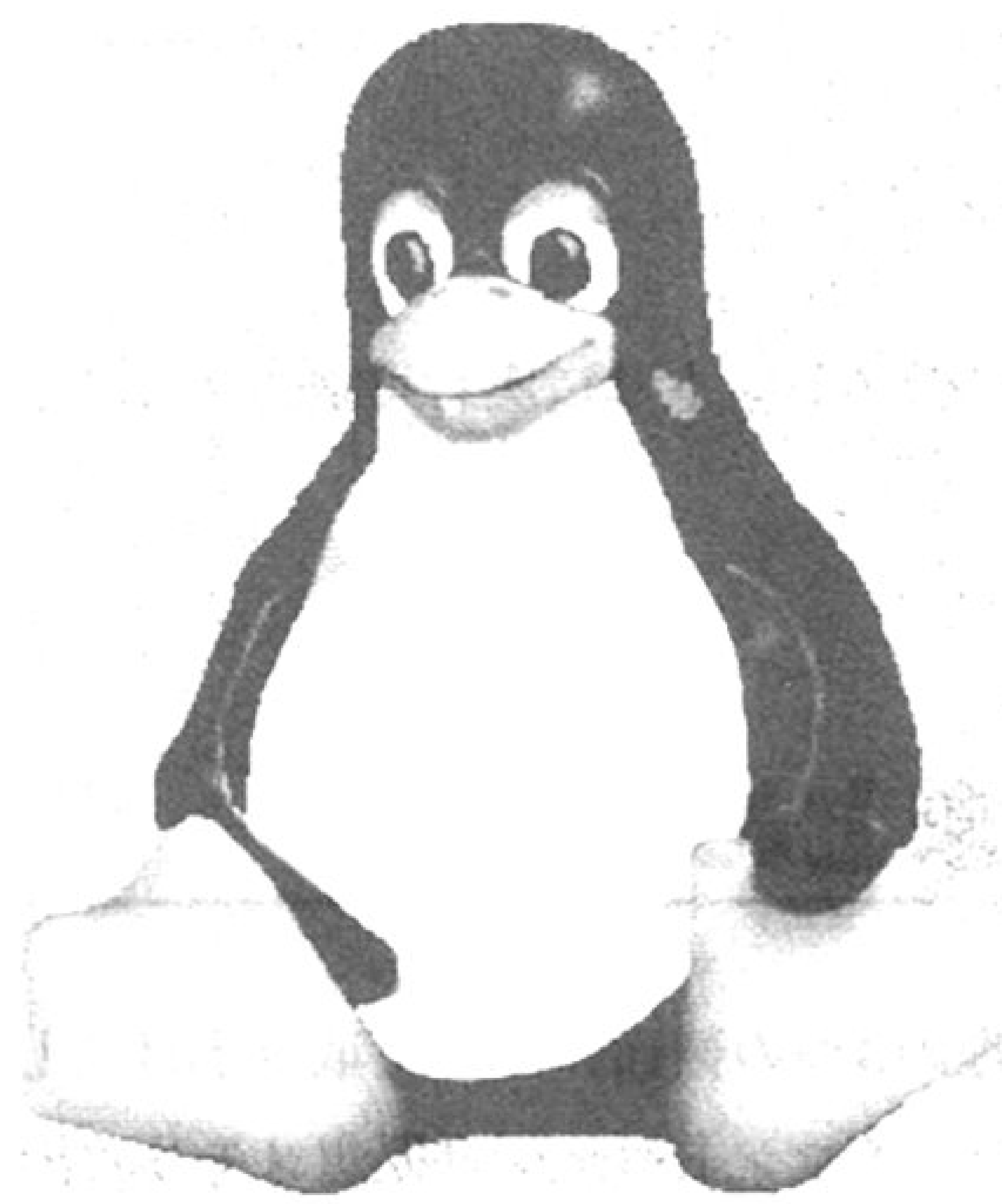
問題がなければ、現在のディレクトリに実行可能ファイル`application`が作成されます。

7.5 この章のまとめ

この章では、データ管理の3つの側面について取り上げました。まず、UNIXのメモリ管理システムについて学びました。デマンドページド仮想メモリは内部的にはかなり高度で複雑な実装になっていますが、メモリの使い方自体はきわめて簡単です。また、UNIXのメモリ管理システムが、どのような方法でオペレーティングシステムとプログラムを不正なメモリアクセスから保護しているかについても説明しました。

次に、ファイルのロックを取り上げ、複数のプログラムが互いに協調してデータにアクセスする方法について説明しました。最初に単純なバイナリセマフォ方式について学び、次に1つのファイルのさまざまな部分を共有アクセスや排他アクセスでロックする方法を学びました。

最後に`dbm`ライブラリを取り上げ、インデックスを使って任意のデータブロックを格納したり、効率的に取得したりする方法を学びました。



Linux

第8章

開発ツール

この章では、UNIXシステムでのプログラム開発に利用できる各種のツールについて説明します。プログラムを開発するにはコンパイラやデバッガが不可欠ですが、UNIXでは、このほかにも非常にたくさんの開発ツールを利用することができます。

UNIXの哲学についてはすでに何度も触れましたが、開発ツールの場合にも、単純な作業を実行する小さなツールを適切に組み合わせることで複雑な作業を実行できます。この章では、いくつかの重要なツールを取り上げ、これらを組み合わせて問題解決に役立てる方法を示します。取り上げる項目は次のとおりです。

- makeコマンドとmakefile
- RCSを使ったソースコード管理
- マニュアルページの記述
- patchとtarを使ったソフトウェアの配布

8.1 makeコマンドとmakefile

ソースファイル2つとヘッダーファイル1つから構成されるようなごく小さなプログラムの場合には、次のように単純にすべてのファイルを再コンパイルすれば、アプリケーションを再構築することができます。

```
$ gcc -o myapp source1.c source2.c
```

しかし、こうした単純なアプローチでは、ソースファイルの数が増えたときにさまざまな問題が生じます。

プログラムのサイズやハードウェアの性能にもよりますが、通常、ソースファイルがたくさんあると、コードを編集してコンパイルし、テストするという手順に相当な時間がかかるようになります。忍耐強いプログラマでも、変更したファイルが1つだけの場合には、すべてのファイルの再コンパイルは避けたいと思うでしょう。

また、ヘッダーファイルが複数あり、これらをいろいろなソースファイルでインクルードしている場合にもやっかいな問題が起きます。たとえば、ヘッダーファイル`a.h`、`b.h`、`c.h`と、Cソースファイル`main.c`、`2.c`、`3.c`があるとします。

Cソースファイルでは、次のようにヘッダーファイルをインクルードしています。

```
/* main.c */
#include "a.h"
...
```

```
/* 2.c */
#include "a.h"
#include "b.h"
...

/* 3.c */
#include "b.h"
#include "c.h"
...
```

c.hを変更した場合、main.cと2.cは再コンパイルする必要はありません。これらのファイルでは、変更されたヘッダーファイルをインクルードしていないからです。つまり、main.cと2.cは、c.hに"依存"していません。一方、ファイル3.cはc.hに依存しているので、c.hを変更した場合には3.cを再コンパイルする必要があります。

この例で、b.hが変更されているにもかかわらず、2.cを再コンパイルしなかったとしたらどうなるでしょうか。おそらく、作成されたプログラムは正しく動作しないでしょう。この場合には、相当な時間を割いて、ソースファイルを正しくコンパイルしていれば発生しなかった問題を突き止めなければなりません。

makeユーティリティを使うと、変更によって影響を受けるファイルを確実に再コンパイルすることができるので、こうした問題をすべて解決できます。

8.1.1 makeの考え方

あとで見るように、makeコマンドにはあらかじめ数多くのルール（規則）が組み込まれていますが、コマンドそれ自体は、基本的にアプリケーションを構築する方法を知りません。このため、makefileというファイルを用意し、アプリケーションを構築する方法をmakeに教えてやる必要があります。

通常、makefileは、ほかのソースファイルと同じディレクトリに置かれます。makefileは、1つのマシン上に同時に複数用意することができます。実際、大きなプロジェクトでは、プロジェクトの各部分に対して個別にmakefileを用意することがあります。

makeコマンドとmakefileを組み合わせて使うと、強力なプロジェクト管理ツールになります。makeコマンドは、ソースコードのコンパイルの管理だけなく、マニュアルページの作成やターゲットディレクトリへのアプリケーションのインストールといった用途にも使用できます。

8.1.2 makefileの構文

makefileを構成するのは、一連の**依存関係**と**ルール**です。依存関係は、ターゲット（作成すべきファイル）と、ターゲットが依存している一連のソースファイルから構成されます。ルールは、ターゲットが依存しているファイルからターゲットを作成する方法を記述したものです。一般に、ターゲットは1つの実行可能ファイルです。

`make`コマンドは、makefileを読み取ってターゲットファイル、つまり作成する必要のあるファイルを決定します。次に、ソースファイルの日付と時刻を比較し、ターゲットを作成するのに必要なルールはどれかを特定します。最終的なターゲットを作成するには、中間的なターゲットの作成が必要になることもあります。この場合、`make`コマンドはmakefileに基づいて、ターゲットを作成する順序と、呼び出すべきルールに関する正しい手順を決定します。

8.1.3 makeのオプションとパラメータ

`make`プログラム自体にも数多くのオプションがあります。このうち、最もよく使われるのは次の3つです。

- **`-k`** 問題が検出された時点で処理を中止するのではなく、エラーが見つかっても処理を続行します。コンパイルに失敗するソースファイルを見つける場合などに便利です。
- **`-n`** 実際には処理を行わず、処理の内容だけを表示します。
- **`-f filename`** makefileとして使うファイルを指定します。このオプションが指定されていない場合、`make`コマンドは、まず現在のディレクトリで`makefile`という名前のファイルを探します。このファイルが存在しなかった場合には、`Makefile`という名前のファイルを探します。UNIXプログラマは、`Makefile`を使うことが多いようです。

特定のターゲット（通常は実行可能ファイル）を作成するには、その名前をパラメータとして`make`コマンドに渡します。ターゲットの指定がない場合、`make`コマンドはmakefile内の最初のターゲットの作成を試みます。多くのプログラマは、`all`をmakefileの最初のターゲットに指定し、その他のターゲットを`all`と依存関係を持つものとして記述しています。この方法を使うと、ターゲットの指定がない場合にデフォルトで作成されるターゲットがわかりやすくなるので、自分でmakefileを記述するときもこの方法に従うとよいでしょう。

◆ 依存関係

依存関係は、最終的なアプリケーションを構成する各ファイルがソースファイルとどのような関係にあるかを表すものです。前に示した例では、最終的なアプリケーションは`main.o`、`2.o`、`3.o`を必要とします（これらに依存しています）。また、`main.o`は`main.c`と`a.h`に、`2.o`は`2.c`と`a.h`と`b.h`に、さらに`3.o`は`3.c`と`b.h`と`c.h`に依存しています。

`main.o`は、`main.c`と`a.h`に対する変更に影響を受けるので、これらのいずれかが変更された場合には`main.c`を再コンパイルし、`main.o`を作成し直す必要があります。

これらのルールをmakefileに記述するには、まず、ターゲットの名前、コロン、スペースまたはタブの順に記述し、そのあとにターゲットを作成するのに使うファイルのリストをスペースまたはタブで区切って指定します。具体的には、次のように記述します。

```
myapp: main.o 2.o 3.o

main.o: main.c a.h

2.o: 2.c a.h b.h

3.o: 3.c b.h c.h
```

このmakefileは、`myapp`が`main.o`、`2.o`、`3.o`に依存していること、`main.o`が`main.c`と`a.h`に依存していることなどを表しています。

これら一連の依存関係から、ソースファイルが互いにどのような関係にあるかがよくわかります。たとえば、ファイル`b.h`を変更した場合には、`2.o`と`3.o`の両方を再コンパイルする必要があります。また、`2.o`と`3.o`も変更されることになるので、ファイル`myapp`も再構築する必要があります。

複数のファイルを作成したい場合には、`all`という"フォニーターゲット"（偽ターゲット）を使うことができます。フォニーターゲットとは、実際にはファイルの名前ではないターゲットのことです。たとえば、バイナリファイル`myapp`とマニュアルページ`myapp.1`からアプリケーションが構成されている場合、フォニーターゲットを使えば、次のように記述することができます。

```
all: myapp myapp.1
```

ターゲット`all`が指定されていない場合、`make`はmakefile内で最初に見つかったターゲットを作成します。

◆ ルール

makefileのもうひとつの重要な要素が、ターゲットの作成方法を記述したルールです。前に示した例で、`make`コマンドが`2.o`を再コンパイルする必要があると判断した場合、実際にはどのようなコマンドが使われるのでしょうか。もちろん、`gcc -c 2.c`のような単純なコマンドで十

分に目的を達成することができます（あとで見るように、**make**は多くのデフォルトルールを持っています）。しかし、インクルードファイルを検索するディレクトリを指定したり、デバッグ用にシンボル情報を設定したい場合などもあります。このような場合には、makefileで明示的なルールを指定します。

makefileの構文に関しては、ひとつ注意しなければならないことがあります。それは、スペースとタブの区別です。ルールは、タブで始まる行に記述されていなければなりません。タブの代わりにスペースが使われている場合、ルールとしては認識されません。タブ1つと複数のスペースは画面上ではどちらも同じように見えるうえに、UNIXプログラミングではスペースとタブを区別しないケースがほとんどなので、makefileでのスペースとタブの区別には注意する必要があります。ルールを記述する行の先頭にタブしか使用できないのは歴史の偶然ですが、既存のmakefileがあまりにも多いために修正するわけにもいかず、現在でもそのままになっています。なお、タブがないために**make**コマンドが失敗する場合、通常はエラーメッセージから原因を特定することができます。

例題 簡単なmakefile

ほとんどのルールは単純なコマンドから構成されており、コマンドラインで入力する内容と同じです。たとえば、前に示した例の場合、ルールを記述した**Makefile1**は次のようになります。

```
myapp: main.o 2.o 3.o
	gcc -o myapp main.o 2.o 3.o

main.o: main.c a.h
	gcc -c main.c

2.o: 2.c a.h b.h
	gcc -c 2.c

3.o: 3.c b.h c.h
	gcc -c 3.c
```

このmakefileの場合、名前は**makefile**でも**Makefile**でもないので、**make**コマンドを呼び出すときには**-f**オプションを使います。さっそく試してみましょう。**Makefile1**だけが置かれたディレクトリで実行します。

```
$ make -f Makefile1
make: *** No rule to make target `main.c', needed by `main.o'.  Stop.
$
```

エラーになりました。なにが原因でしょうか。**make**コマンドはmakefile内の最初のターゲット、つまり**myapp**を作成しようとします。このとき、ほかの依存関係を調べて**main.c**が必要であるという結論を得ます。しかし、**main.c**というファイルはまだ作成していませんし、**main.c**

を作成するための情報もmakefileには記述していません。このためmakeコマンドはエラーを報告したのです。

では、ソースファイルを作成してもう一度やり直してみましょう。ここではmakeコマンドの動作を確かめることが目的なので、単純にtouchコマンドで空のヘッダーファイルを作成します。

```
$ touch a.h
$ touch b.h
$ touch c.h
$
```

main.cにはmain関数を記述し、この中でfunction_twoとfunction_threeを呼び出します。ファイル2.cと3.cでは、それぞれfunction_twoとfunction_threeを定義します。また、これら3つのCソースファイルでは適切なヘッダーファイルをインクルードします。完成したファイルの内容は次のようになります。

```
/* main.c */
#include "a.h"

extern void function_two();
extern void function_three();

int main()
{
    function_two();
    function_three();
    return (0);
}

/* 2.c */
#include "a.h"
#include "b.h"

void function_two() {
}

/* 3.c */
#include "b.h"
#include "c.h"

void function_three() {
}
```

もう一度makeを実行します。

```
$ make -f Makefile1
gcc -c main.c
gcc -c 2.c
```

```
gcc -c 3.c
gcc -o myapp main.o 2.o 3.o
$
```

今度は成功しました。

解説

makeコマンドは、makefileの依存関係の部分を処理し、作成する必要のあるファイルとその順序を決定します。makefileでは**myapp**の作成方法が一番最初に記述されていますが、**make**コマンドはファイルを作成する順序を正しく決定しています。次に、ファイルを作成するためのルールの部分で指定されたコマンドを実行します。**make**コマンドは、実行と同時にコマンドを表示します。

では、ファイル**b.h**に加えた変更が正しく処理されるかどうか調べてみましょう。

```
$ touch b.h
$ make -f Makefile1
gcc -c 2.c
gcc -c 3.c
gcc -o myapp main.o 2.o 3.o
$
```

うまくいきました。**make**コマンドは、makefileを読み取り、**myapp**を再構築するのに必要な最小限のコマンドだけを正しい順序で実行しています。

オブジェクトファイルを削除するとどうなるでしょうか。

```
$ rm 2.o
$ make -f Makefile1
gcc -c 2.c
gcc -o myapp main.o 2.o 3.o
$
```

makeコマンドは、必要な処理を正しく判断しています。

8.1.4 makefile内のコメント

makefileにはコメントを入れることができます。makefileでは、1つの**#**から行の終わりまでがコメントです。Cソースファイルの場合と同様、makefileにコメントを入れておくと、ファイル作成時の意図をわかりやすく伝えることができます。

8.1.5 makefileのマクロ

これまで見てきたように、makeとmakefileは、複数のソースファイルを扱うプロジェクトを管理するための非常に強力なツールです。しかし、ファイルの数が非常に多いプロジェクトではmakefileも大がかりなものになり、管理の柔軟性が損なわれる可能性があります。このような場合には、マクロを使ってより一般的な形式でmakefileを記述することができます。

makefileのマクロは、`MACRONAME=value`という形で定義し、`$(MACRONAME)`または`${MACRONAME}`でその値にアクセスします（makeのバージョンによっては`$MACRONAME`を使用することもできます）。`=`以降、行の終わりまでを空白にすれば、名前を空にしておくことができます。

makefileでは、しばしばマクロを使ってコンパイラのオプションを指定します。たとえば、アプリケーションの開発中には、最適化を行わずにデバッグ情報を含めてコンパイルすることが多いはずです。一方、リリースバージョンを作成するときには、高速に実行できるサイズの小さなバイナリを作成するために、デバッグ情報を含めずに最適化を行うほうが一般的です。こうしたケースでは、マクロを使ってコンパイラのオプションを指定すると便利です。

また、前に例として取り上げた`Makefile1`では、コンパイラの名前が`gcc`であると仮定しています。しかし、ほかのUNIXシステムでは、`gcc`ではなく`cc`や`c89`が使われている可能性もあります。作成したmakefileを異なるバージョンのUNIXで使いたい場合や、同じシステム上でも別のコンパイラを使用する場合、コンパイラの名前が`gcc`になっているmakefileでは、コンパイラの指定をすべて修正する必要があります。このような場合には、システムに依存する部分をマクロを使って1箇所にまとめておけば変更も容易です。

通常、マクロはmakefile自体の中で定義しますが、`make CC=c89`のように、`make`コマンドを呼び出すときにマクロ定義を指定することもできます。コマンドラインで指定した定義はmakefileでの定義より優先されます。

例題 マクロを使ったmakefile

次に示す`Makefile2`は、マクロを使って`Makefile1`を書き換えたものです。

```
all: myapp

# Which compiler
CC=gcc

# Where are include files kept
INCLUDE=.

# Options for development
CFLAGS=-g -Wall -ansi

# Options for release
```

```
# CFLAGS=-O -Wall -ansi

myapp: main.o 2.o 3.o
        $(CC) -o myapp main.o 2.o 3.o

main.o: main.c a.h
        $(CC) -I$(INCLUDE) $(CFLAGS) -c main.c

2.o: 2.c a.h b.h
        $(CC) -I$(INCLUDE) $(CFLAGS) -c 2.c

3.o: 3.c b.h c.h
        $(CC) -I$(INCLUDE) $(CFLAGS) -c 3.c
```

`Makefile1`で作成したアプリケーションとオブジェクトファイルを削除したあと、新しいmakefileを指定して`make`を実行します。出力は次のようになります。

```
$ rm *.o myapp
$ make -f Makefile2
gcc -I. -g -Wall -ansi -c main.c
gcc -I. -g -Wall -ansi -c 2.c
gcc -I. -g -Wall -ansi -c 3.c
gcc -o myapp main.o 2.o 3.o
$
```

解説

`make`コマンドは、Cコンパイラが`#define`を置き換えるのと同様に、`$(CC)`、`$(CFLAGS)`、`$(INCLUDE)`を適切な定義で置き換えます。このため、コンパイルコマンドを変更する場合でもわずか1箇所を変更するだけですみます。

8.1.6 複数のターゲット

makefileでは、複数のターゲットを作成できるようにしたり、一連のコマンドを1つにまとめたりすることもできます。さっそく、不要なオブジェクトを削除するターゲット`clean`と、指定したディレクトリにアプリケーションをインストールするターゲット`install`を追加してみましょう。

例題 複数のターゲット

```
all: myapp

# Which compiler
CC=gcc

# Where to install
INSTDIR=/usr/local/bin

# Where are include files kept
INCLUDE=.

# Options for development
CFLAGS=-g -Wall -ansi

# Options for release
# CFLAGS=-O -Wall -ansi


myapp: main.o 2.o 3.o
	$(CC) -o myapp main.o 2.o 3.o

main.o: main.c a.h
	$(CC) -I$(INCLUDE) $(CFLAGS) -c main.c

2.o: 2.c a.h b.h
	$(CC) -I$(INCLUDE) $(CFLAGS) -c 2.c

3.o: 3.c b.h c.h
	$(CC) -I$(INCLUDE) $(CFLAGS) -c 3.c


clean:
	-rm main.o 2.o 3.o

install: myapp
	@if [ -d $(INSTDIR) ]; \
	   then \
	   cp myapp $(INSTDIR);\
	   chmod a+x $(INSTDIR)/myapp;\
	   chmod og-w $(INSTDIR/myapp;\
	   echo "Installed in $(INSTDIR)";\
	else \
	   echo "Sorry, $(INSTDIR) does not exist";\
	fi
```

このmakefileにはいくつか注目すべき点があります。まず、特別なターゲット`all`では、これまで同様に**myapp**をターゲットとして指定しています。このため、ターゲットを指定せずに**make**を実行すると、デフォルトでターゲット**myapp**が構築されます。

次に重要な点は、cleanとinstallの2つのターゲットが追加されていることです。ターゲットcleanでは、rmコマンドを使ってオブジェクトを削除します。コマンドの前に付いている-は、makeに対してコマンドの結果を無視するように指示するためのものです。このため、オブジェクトが存在しない場合やrmコマンドがエラーを返した場合でも、make cleanは成功します。clean:のあとは空白なので、ターゲットcleanはどのファイルにも依存していません。したがって、cleanは常に古いと判断され、ターゲットにcleanが指定された場合には、常にそのルールが実行されます。

ターゲットinstallはmyappに依存しているので、installを作成するときには、その前にほかのコマンドを実行する必要があります。installを作成するためのルールは、いくつかのシェルスクリプトコマンドから構成されています。makeはルールを実行するときにシェルを呼び出しますが、1つのルールにつき新しいシェルを1つ呼び出すので、バックスラッシュを使ってすべてのスクリプトコマンドを1つの論理行に収め、シェルを1回呼び出すだけでルールを実行できるようにしています。コマンドの先頭に付いている@記号は、ルールを実行するときにコマンドを標準出力に表示しないようにするためのものです。

新しく作成したコマンドは、通常のユーザーのアクセス権では/usr/local/binにインストールすることはできません。インストール先のディレクトリのパーミッションを変更するか、またはsuコマンドを使ってスーパーユーザーになってからmake installを実行します。

```
$ rm *.o myapp
$ make -f Makefile3
gcc -I. -g -Wall -ansi -c main.c
gcc -I. -g -Wall -ansi -c 2.c
gcc -I. -g -Wall -ansi -c 3.c
gcc -o myapp main.o 2.o 3.o
$ make -f Makefile3
make: Nothing to be done for `all'.
$ rm myapp
$ make -f Makefile3 install
gcc -o myapp main.o 2.o 3.o
Installed in /usr/local/bin
$ make -f Makefile3 clean
rm main.o 2.o 3.o
$
```

解説

まず、myappとすべてのオブジェクトを削除します。次に、makeコマンドを実行します。allがターゲットとして選択され、myappが作成されます。次にもう一度makeを実行します。myappは最新の状態になっているので、makeはなにも行いません。今度は、ファイルmyappを削除してからmake installを実行します。myappが再作成され、インストール先のディレクトリにコピーされます。最後にmake cleanを実行します。この操作によってオブジェクトが削除されます。

8.1.7 ビルトインルールとサフィックスルール

これまでに取り上げたmakfileでは、各ステップの実行方法をそのまま記述していました。実際には、**make**は数多くの**ビルトインルール**を持っており、これらのルールを使うことでmakefileの記述を大幅に簡略化できます。

◆ ビルトインルール

多くのルールは同じです。このため、**make**は数多くのビルトインルールを持っています。例を示しましょう。次のコードは、Hello Worldと表示するおなじみのプログラム、**foo.c**です。

```
#include <stdlib.h>
#include <stdio.h>

int main()
{
    printf("Hello World\n");
    exit (EXIT_SUCCESS);
}
```

makefileを指定せずに**make**を使ってコンパイルします。

```
$ make foo
cc     foo.c   -o foo
$
```

gccではなくccが呼び出されていますが、この実行例からわかるように、**make**はコンパイラの呼び出し方を知っています。こうしたビルトインルールは、**推論ルール**と呼ばれることもあります。デフォルトのルールではマクロが使われているので、次の例のようにマクロに新しい値を指定すれば、デフォルトの動作を変更することもできます。

```
$ rm foo
$ make CC=gcc CFLAGS="-Wall -g" foo
gcc -Wall -g    foo.c   -o foo
$
```

-pオプションを指定して**make**を実行すると、ビルトインルールを表示させることができます。次の例は、GNUバージョンの**make**での実行例です。ビルトインルールは非常に数が多いので、ここではその一部だけを示しています。

```
OUTPUT_OPTION = -o $@

COMPILE.c = $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
```

```
.c.o:
        $(COMPILE.c) $< $(OUTPUT_OPTION)
```

ビルトインルールを活用し、オブジェクトを作成するルールを省略して依存関係だけを指定すると、**myapp**を作成するためのmakefileの該当部分は、次のように簡潔に記述することができます。

```
main.o: main.c a.h
2.o: 2.c a.h b.h
3.o: 3.c b.h c.h
```

◆ サフィックスルール

上に示したビルトインルールは、**サフィックス**（DOSの拡張子に相当）によって機能しています。つまり、**make**は、あるサフィックスを持つファイルから別のサフィックスを持つファイルを作成する際、使用できるルールはどれかを知っています。もっともよく使われるルールは、**.c**で終わるファイルから**.o**で終わるファイルを作成するためのルールです。このルールでは、Cコンパイラを使ってソースファイルのコンパイルを行います（リンクは行いません）。

場合によっては、新しいルールの作成が必要になることがあります。筆者は、同じソースファイルをコンパイルするときに複数のコンパイラを使うことがあります。具体的には、MS-DOSではコンパイラを2つ使用し、Linuxでは**gcc**を使っています。たとえば、ソースファイルがCではなくC++で記述されている場合、MS-DOSのコンパイラではサフィックスに**.cpp**を使う必要があります。しかし、Linuxで使う**make**には、**.cpp**ファイルをコンパイルするためのビルトインルールがありません（UNIXではC++のファイル拡張子に**.cc**を使うほうが一般的なので、**.cc**をコンパイルするルールは用意されています）。

このような場合には、ソースファイルごとにルールを指定するか、または拡張子**.cpp**を持つファイルからオブジェクトを作成する新しいルールを**make**に教えてやる必要があります。プロジェクトを構成するソースファイルがたくさんある場合には、新しいルールを指定すれば、ソースファイルごとにルールを入力する手間を省くことができ、新しいソースファイルも簡単に追加できます。

新しいサフィックスルールを追加するには、新しいサフィックスを**make**に教えるための行をmakefileの中に追加します。次に、新しいサフィックスを使ってルールを記述します。このとき、**.<old suffix>.<new suffix>:**という特別な構文を使います。この構文によって、古いサフィックスを持つファイルからベース名が同じで新しいサフィックスを持つファイルを作成するための一般的なルールが定義されます。

例題 サフィックスルール

次に示すのは、**.cpp**ファイルから**.o**ファイルを作成するための新しい一般的なルールを追加したmakefileの一部です。この部分は、ファイルの先頭の**all: myapp**の行の直後に置きます。makefileの名前は**Makefile5**とします。

```
.SUFFIXES:      .cpp

.cpp.o:
        $(CC) -xc++ $(CFLAGS) -I$(INCLUDE) -c $<
```

新しいルールを試してみます。

```
$ cp foo.c bar.cpp
$ make -f Makefile5 bar
gcc -xc++ -g -Wall -ansi -I. -c bar.cpp
gcc   bar.o   -o bar
rm bar.o
$
```


解説

特別な依存関係**.cpp.o:**は、そのあとのルールでサフィックス**.cpp**を持つファイルからサフィックス**.o**を持つファイルを作成する方法が記述されていることを示します。

この依存関係を記述した時点ではファイルの実際の名前はまだわからないので、ルールでは特別のマクロ名を使っています。このうち**$<**は、出発点となるファイル名（古いサフィックスを持つファイル名）を表します。ルールでは、**.cpp**ファイルから**.o**ファイルを作成する方法だけを指定しています。これは、**make**がオブジェクトファイルからバイナリの実行可能ファイルを作成する方法をすでに知っているからです。

サフィックスルールを追加したmakefileを指定して**make**を実行すると、新しいルールが適用されて**bar.cpp**から**bar.o**が作成され、次にビルトインルールによって**.o**ファイルから実行可能ファイルが作成されます。**-xc++**は、**gcc**にC++ソースファイルであることを知らせるためのフラグです。

上のサンプルmakefileで追加したルールは非常に簡単なもので、実際にはもう少し複雑なルールを記述することになるでしょう。しかし、サフィックスルールの使い方については十分理解できたと思います。次に示すのは、よく使う特殊なマクロです。

- **$@**　現在のターゲットのフルネーム
- **$?**　現在のターゲットが依存していて、現在のターゲットよりも新しいファイルのリスト
- **$<**　ターゲットに依存していて、ターゲットよりも新しい単一のファイル
- **$***　サフィックスを除いたターゲットファイルの名前

8.1.8 makeを使ったライブラリの管理

大規模なプロジェクトでは、コンパイルの生成物を**ライブラリ**として管理すると便利なことがあります。一般に拡張子.a（aは**アーカイブ**の意）を持つこれらのファイルは、オブジェクトファイルの集合です。makeコマンドにはライブラリを扱うための特別な構文があり、この構文を使うことでライブラリを非常に簡単に管理できます。

具体的には、lib(file.o)のように指定します。これは、ライブラリlib.aに格納されているオブジェクトファイルfile.oという意味です。makeコマンドは、ライブラリを管理するための次のようなビルトインルールを持っています。

```
.c.a:
        $(CC) -c $(CFLAGS) $<
        $(AR) $(ARFLAGS) $@ $*.o
```

通常、マクロ$(AR)と$(ARFLAGS)は、デフォルトでそれぞれコマンドarとオプションrvに展開されます。ルールは2行で記述されており、.cファイルから.aライブラリを作成するには、ルールを2つ適用しなければならないことを示しています。

最初のルールは、ソースファイルをコンパイルしてオブジェクトを作成する必要があることを示しています。2番目のルールは、arコマンドを使って新しいオブジェクトファイルを追加し、ライブラリを更新する必要があることを示しています。たとえば、fudというライブラリにbas.oというファイルが含まれているとします。この場合、最初のルールでは$<がbas.cに置き換えられます。また、2番目のルールでは$@がライブラリfud.aに、$*がbasに展開されます。

例題 ライブラリの管理

ライブラリを扱う構文の使い方は実際には簡単です。これまでの例で使ったアプリケーションを変更し、ファイル2.oと3.oをmylib.aという名前のライブラリで管理してみましょう。makefileに変更を加え、Makefile6を作成します（前の例で追加したC++プログラムを処理するためのルールは削除します）。

```
all: myapp

# Which compiler
CC=gcc

# Where to install
INSTDIR=/usr/local/bin

# Where are include files kept
INCLUDE=.
```

```
# Options for development
CFLAGS=-g -Wall -ansi

# Options for release
# CFLAGS=-O -Wall -ansi

# Local Libraries
MYLIB=mylib.a

myapp: main.o $(MYLIB)
	$(CC) -o myapp main.o $(MYLIB)


$(MYLIB): $(MYLIB)(2.o) $(MYLIB)(3.o)
main.o: main.c a.h
2.o: 2.c a.h b.h
3.o: 3.c b.h c.h

clean:
	-rm main.o 2.o 3.o $(MYLIB)

install: myapp
	@if [ -d $(INSTDIR) ]; \
	   then \
	   cp myapp $(INSTDIR);\
	   chmod a+x $(INSTDIR)/myapp;\
	   chmod og-w $(INSTDIR/myapp;\
	   echo "Installed in $(INSTDIR)";\
	else \
	   echo "Sorry, $(INSTDIR) does not exist";\
	fi
```


新しいmakefileを使って**make**コマンドを実行します。

```
$ rm -f myapp *.o mylib.a
$ make -f Makefile6
gcc -g -Wall -ansi   -c main.c -o main.o
gcc -g -Wall -ansi   -c 2.c -o 2.o
ar rv mylib.a 2.o
a - 2.o
gcc -g -Wall -ansi   -c 3.c -o 3.o
ar rv mylib.a 3.o
a - 3.o
gcc -o myapp main.o mylib.a
$ touch c.h
$ make -f Makefile6
gcc -g -Wall -ansi   -c 3.c -o 3.o
ar rv mylib.a 3.o
r - 3.o
gcc -o myapp main.o mylib.a
$
```

解説

まず、ライブラリとオブジェクトをすべて削除し、makeコマンドがmyappを作成できるようにします。makeコマンドを実行すると、ソースファイルのコンパイルとライブラリの作成が行われ、main.oとライブラリがリンクされてmyappが作成されます。次に、3.oの依存関係をテストするために、touchコマンドを使ってc.hを更新します。もう一度makeを実行すると、3.cが正しく再コンパイルされてライブラリが更新され、最後に新しいmyapp実行可能ファイルが作成されます。

8.1.9 makefileとサブディレクトリ

大規模なプロジェクトでは、ライブラリを構成するファイルをサブディレクトリに置き、メインのファイルと区別しておくと便利なことがあります。

このような場合にmakeを使って作業を効率化するには、2つの方法があります。ひとつは、サブディレクトリにもうひとつmakefileを用意する方法です。このmakefileでは、ファイルのコンパイル、ライブラリへの格納、1つ上のメインディレクトリへのライブラリのコピーを行えるようにしておきます。上位ディレクトリのメインのmakefileには、ライブラリを作成するためのルールを記述し、このルールでサブディレクトリのmakefileを呼び出すようにします。

```
mylib.a:
        (cd mylibdirectory;$(MAKE))
```

このルールは、常にmylib.aの作成を試みなければならないことを意味しています（依存関係を持たないターゲットは常に古いと判断されます)。makeは、ライブラリを作成するルールを呼び出すときに、サブディレクトリmylibdirectoryに移動し、そこで新しくmakeコマンドを呼び出してライブラリを作成します。このとき、新しいシェルが呼び出されるので、makefileを使うプログラム自体はcdを実行しません。ライブラリ作成のルールを実行するために呼び出されたシェルだけがサブディレクトリで実行されます。ルールをカッコで囲んでいるのは、すべてのコマンドを単一のシェルで実行するためです。

もうひとつの方法は、makefileにいくつかのマクロを追加する方法です。具体的には、ディレクトリを表すDとファイル名を表すFを使って、ビルトインサフィックスルールの.c.oを次のルールで上書きします。

```
.c.o:
        $(CC) $(CFLAGS) -c $(@D)/$(<F) -o $(@D)/$(@F)
```

このルールでは、サブディレクトリ内のファイルをコンパイルしたあと、作成したオブジェクトをそのままサブディレクトリに置いておきます。次に、現在のディレクトリにライブラリを作成するために、次のような依存関係とルールを記述します。

```
mylib.a:   mydir/2.o mydir/3.o
	ar -rv mylib.a $?
```

どちらの方法を使うべきかはプロジェクトに応じて決めるとよいでしょう。サブディレクトリを使わないプロジェクトではソースディレクトリに非常に多くのファイルが置かれることになりますが、ここで説明している方法でmakeを利用すれば、わずかな変更でサブディレクトリを効率的に活用することができます。

8.1.10 GNU makeとgcc

GNUのmakeとgccを使っている場合には、いくつかの便利なオプションを使うことができます。

まずひとつは、makeの-jNオプションです（jはジョブの意）。このオプションを指定すると、makeコマンドは同時にN個のコマンドを実行できるようになります。プロジェクトの各部分を個別にコンパイルすることが可能な場合には、このオプションを指定することで複数のルールを同時に処理できます。システム構成にもよりますが、非常に多くのソースファイルがある場合には、この方法を使うと再コンパイルの時間を大幅に短縮することができます。一般には、-j3程度の小さな値から試してみるとよいでしょう。ただし、マシンを使っているユーザーがほかにもいる場合には、-jオプションを慎重に使わなければなりません。コンパイルするたびに多数のプロセスが起動されると、ほかのユーザーの作業に悪影響を及ぼす可能性があるからです。

もうひとつの便利なオプションは、gccの-MMオプションです。このオプションを指定すると、makeで使用できる形式で依存関係のリストが生成されます。プロジェクトを構成するソースファイルの数が多く、各ソースファイルでさまざまなヘッダーファイルをインクルードしている場合には、正しい依存関係の把握が困難になることがあります。ひとつのやり方として、各ソースファイルですべてのヘッダーファイルをインクルードすることもできますが、これでは不要なコンパイル作業が増えてしまいます。一方、必要な依存関係を記述し忘れると、再コンパイルの必要なファイルがコンパイルされないなどの重大な問題が発生します。gccの-MMオプションを使えば、こうした問題を避けることができます。

例題 gcc -MM

```
$ gcc -MM main.c 2.c 3.c
main.o: main.c a.h
2.o: 2.c a.h b.h
3.o: 3.c b.h c.h
$
```

解説

gccコンパイラは、makefileに挿入できる形式で必要な依存関係を出力します。あとは、出力を一時ファイルなどに保存し、これをmakefileに挿入するだけです。

> メモ
> -MMオプションと同じような機能を持つツールとしてmakedependがあります。makedependは、指定されたmakefileの最後に依存関係を追加します。

8.2 ソースコード管理

プロジェクトが複雑になってくると、ソースファイルの変更を管理することが重要な意味を持つようになります。特に、複数のユーザーが作業するプロジェクトでは、ソースファイルの変更管理は非常に大切です。

UNIXでは、**RCS**（Revision Control System）と**SCCS**（Source Code Control System）の2つのソースファイル管理システムが広く使われています。RCSユーティリティは、ソースファイルも含めてFree Software Foundationから公開されています。SCCSはAT&TによってSystem VバージョンのUNIXに導入され、現在ではX/Open仕様に含まれています。また、サードパーティからも多数のソースコード管理システムが提供されています。このうち、最もよく知られているものに**CVS**（Concurrent Version System）があります。CVSはSCCSやRCSより高機能ですが、まだ広く使われるまでには至っていないようです。

ここでは、まずRCSについて重点的に説明したあと、RCSのコマンドとSCCSのコマンドを比較します。RCSとSCCSのコマンドは互いによく似た機能を持っているので、RCSからSCCSへ、またSCCSからRCSへはそれほど苦労せずに移行することができます。RCSにはMS-DOSで利用できるバージョンもあり、商業ベースのサポートを盛り込んだ製品も出ています。

8.2.1 RCS

RCSシステムは、ソースファイルを管理するための一連のコマンドから構成されています。RCSシステムでは、以前のバージョンを再作成するのに十分なレベルの変更内容のリストを記録し、これを単一のファイルで保守することによって、変更が加えられたソースファイルを追跡します。また、変更内容ごとにコメントを付けることができるので、あとでファイルに対する変更の履歴をたどることができます。

実際にプロジェクトで使うときには、プロジェクトの進行に合わせてソースファイルに対する重要な変更やバグフィクスを記録し、変更を加えるごとにコメントを付けることができます。こ

の機能は、ファイルに対する変更をあとから検討したり、バグがいつ解消されたか、またはバグがいつ混入したかを特定する場合に非常に便利です。

◆ rcsコマンド

RCSでは、まず管理するファイルの初期バージョンを作成します。ここでは、以前に例として使った**foo.c**の冒頭に次のようなコメントを挿入した**important.c**を使って説明します。

```
/*
  This is an important file for managing this project.
  It implements the canonical "Hello World" program.
*/
```

最初に必要になる作業は、**rcs**コマンドを使ってファイルに対するRCS管理を初期化することです。コマンド**rcs -i**を実行すると、RCS管理ファイルが初期化されます。

```
$ rcs -i important.c
RCS file: important.c,v
enter description, terminated with single '.' or end of file:
NOTE: This is NOT the log message!
>> This is an important demonstration file
>> .
done
$
```

コメントは複数の行にわたってもかまいません。コメントを終えるには、コメント入力用のプロンプトに対してピリオド(**.**)を1つだけ入力するか、またはファイルの終わりを表す文字(通常は**Ctrl-D**)を入力します。

コマンドを実行すると、次のように**,v**という拡張子を持つ読み取り専用ファイルが新しく作成されます。

```
$ ls -l
-rw-r--r--   1 rick     users        226 Feb 11 16:25 important.c
-r--r--r--   1 rick     users        105 Feb 11 16:26 important.c,v
$
```

RCSが管理するファイルを別のディレクトリに置きたい場合には、**rcs**を初めて実行する前に、**RCS**という名前のサブディレクトリを作成します。こうすると、以後すべての**rcs**コマンドは、自動的に**RCS**サブディレクトリを対象として実行されるようになります。

◆ ciコマンド

初期化を済ませたら、現在のバージョンを格納する**ci**コマンドを使ってファイルを**チェックイン**します。

```
$ ci important.c
important.c,v  <--  important.c
initial revision: 1.1
done
$
```

rcs -iを実行していない場合には、ファイルに関する説明の入力を求められます。このようにユーザーの操作エラーに対して比較的寛大な点は、SCCSに対するRCSの利点のひとつです。

ciコマンドの実行後にディレクトリのリストを表示すると、**important.c**が削除されていることがわかります。

```
$ ls -l
-r--r--r--   1 rick     users         442 Feb 11 16:30 important.c,v
$
```

ファイルの内容と管理情報は、すべてRCSファイルの**important.c,v**に格納されています。

◆ coコマンド

ファイルを変更したい場合には、まずファイルを**チェックアウト**しなければなりません。ファイルを読むだけなら、**co**コマンドを使って現在のバージョンを再作成すればすみます。この場合、ファイルのパーミッションは読み取り専用に変更されます。しかし、ファイルを編集する場合には、**co -l**コマンドを使ってファイルをロックする必要があります。

ロックが必要になる理由は、複数のメンバーが参加するプロジェクトではファイルを編集するメンバーが1人だけであることを保証する必要があるからです。ファイルの1つのバージョンの1つのコピーだけが書き込みパーミッションを持つのもそのためです。書き込みパーミッションを設定してファイルをチェックアウトすると、RCSファイルはロックされます。

実際に試してみましょう。

```
$ co -l important.c
important.c,v  -->  important.c
revision 1.1 (locked)
done
$
```

このとき、ディレクトリの内容は次のようになっています。

```
$ ls -l
-rw-r--r--   1 rick     users         226 Feb 11 16:35 important.c
-r--r--r--   1 rick     users         452 Feb 11 16:35 important.c,v
$
```

これで、ファイルを編集し、新しい変更内容を追加することができるようになりました。実際

にファイルを変更し、再びciコマンドを使って変更内容を保存します。important.cの出力部分を次のように変更します。

```
    printf("Hello World\n");
    printf("This is an extra line added later\n");
```

ciコマンドを実行します。

```
$ ci important.c
important.c,v  <--  important.c
new revision: 1.2; previous revision: 1.1
enter log message, terminated with single '.' or end of file:
>> Added an extra line to be printed out.
>> .
done
$
```

メモ

ファイルをチェックインし、ロックをそのままにして編集作業を続けたい場合には、-lオプションを指定してciコマンドを実行します。こうすると、自動的に同じユーザーに対してファイルがチェックアウトされます。

編集後のバージョンを格納すると、important.cは再び削除されます。

```
$ ls -l
-r--r--r--   1 rick     users         633 Feb 11 16:37 important.c,v
$
```

◆ rlogコマンド

ファイルの変更内容の概要を表示したい場合には、rlogコマンドを使います。

```
$ rlog important.c

RCS file: important.c,v
Working file: important.c
head: 1.2
branch:
locks: strict
access list:
symbolic names:
comment leader: " * "
keyword substitution: kv
total revisions: 2;     selected revisions: 2
description:
```

```
This is an important demonstration file
----------------------------
revision 1.2
date: 1998/02/11 16:37:35;  author: rick;  state: Exp;  lines: +1 -0
Added an extra line to be printed out.
----------------------------
revision 1.1
date: 1998/02/11 16:30:19;  author: rick;  state: Exp;
Initial revision
=============================================================================
$
```

最初の部分は、ファイルに関する説明と、**rcs**で使われているオプションです。そのあとには、ファイルのリビジョンと各リビジョンをチェックインしたときに入力したテキストが、最新のものから順にリストアップされています。リビジョン1.2のところにlines: +1 -0という記述があることに注目してください。これは、1行が追加され、0行が削除されたことを示しています。

ファイルの最初のバージョンを取り出したい場合には、必要なリビジョンを指定して**co**コマンドを実行します。

```
$ co -r1.1 important.c
important.c,v  -->  important.c
revision 1.1
done
$
```


メモ

ciコマンドにも**-r**オプションがあります。**-r**オプションを使うと、バージョン番号を指定した値にすることができます。たとえば、次のようにします。

```
ci -r2 important.c
```

このコマンドは、**important.c**をバージョン2.1としてチェックインします。RCSとSCCSのどちらの場合も、最初のマイナーバージョン番号は(0ではなく)1です。

◆ rcsdiffコマンド

2つのリビジョンの間でなにが変更されたのかを知りたい場合には、**rcsdiff**コマンドを使います。

```
$ rcsdiff -r1.1 -r1.2 important.c
===================================================================
RCS file: important.c,v
retrieving revision 1.1
retrieving revision 1.2
diff -r1.1 -r1.2
```

```
11a12
>       printf("This is an extra line added later\n");

$
```

この出力から、11行目のあとに1行が追加されたことがわかります。

◆ リビジョンの特定

RCSでは、ソースファイルに特別な文字列（マクロ）を挿入して変更内容の追跡に役立てることができます。最もよく使われるマクロは、**$RCSfile$**と**Id**の2つです。**$RCSfile$**はファイルの名前に展開され、**Id**はリビジョンを特定する文字列に展開されます。これら以外にサポートされている特別な文字列の一覧については、マニュアルページを参照してください。マクロは、ファイルのリビジョンがチェックアウトされるたびに新たに展開され、リビジョンがチェックインされるときに自動的に更新されます。

ではさっそく、**important.c**を修正してマクロを使ってみましょう。

```
$ co -l important.c
important.c,v  -->  important.c
revision 1.2 (locked)
done
$
```

次のように修正します。

```
/*
  This is an important file for managing this project.
  It implements the canonical "Hello World" program.

  Filename: $RCSfile$

*/

#include <stdlib.h>
#include <stdio.h>

static char *RCSinfo = "$Id$";

int main() {
    printf("Hello World\n");
    printf("This is an extra line added later\n");
    printf("This file is under RCS control. It's ID is\n%s\n", RCSinfo);
    exit (EXIT_SUCCESS);
}
```

このリビジョンをチェックインし、特別な文字列がどのように扱われるかを確認します。

```
$ ci important.c
important.c,v  <--  important.c
new revision: 1.3; previous revision: 1.2
enter log message, terminated with single '.' or end of file:
>> Added $RCSfile$ and $Id$ strings
>> .
done
$
```

ディレクトリにはRCSファイルだけが残ります。

```
$ ls -l
-r--r--r--   1 rick     users         907 Feb 11 16:55 important.c,v
$
```

coコマンドを使ってファイルをチェックアウトし、現在のバージョンのソースファイルを確認すると、マクロが展開されていることがわかります。

```
/*
  This is an important file for managing this project.
  It implements the canonical "Hello World" program.

  Filename: $RCSfile: important.c,v $

*/

#include <stdlib.h>
#include <stdio.h>

static char *RCSinfo = "$Id: important.c,v 1.3 1998/02/11 16:55:04 rick Exp $";

int main() {
    printf("Hello World\n");
    printf("This is an extra line added later\n");
    printf("This file is under RCS control. It's ID is\n%s\n", RCSinfo);
    exit (EXIT_SUCCESS);
}
```

例題 GNU makeとRCSの併用

GNU makeは、RCSファイルを管理するためのビルトインルールをすでに持っています。ここではソースファイルを削除し、不足しているソースファイルがmakeによってどのように扱われるかを確かめてみましょう。

```
$ rm -f important.c
$ make important
co  important.c,v important.c
```

```
important.c,v  -->  important.c
revision 1.3
done
cc    -c important.c -o important.o
cc   important.o   -o important
rm important.o important.c
$
```

解説

makeは、拡張子を持たないファイルを作成するには、同じ名前で拡張子**.c**を持つファイルをコンパイルすればよいことを知っています。また、RCSを使うことで**imortant.c,v**から**important.c**を作成できることも知っています。上の例では、**important.c**というファイルは存在しないので、**make**は**co**コマンドを使って最新のリビジョンをチェックアウトし、**.c**ファイルを作成します。コンパイルが終わるとあとかたづけを行い、**important.c**を削除します。

◆ ident コマンド

identコマンドを使うと、**Id**文字列を含むファイルのバージョンを調べることができます。**important.c**ではこの文字列を変数に格納しているので、コンパイルの結果としてできる実行可能ファイルにも文字列が含まれています。ただし、コンパイラによっては、最適化の際にコード内でアクセスされていない変数を取り除いてしまうことがあります。このような場合には、ダミーのアクセスを行うコードを追加すれば問題を回避することができます（コンパイラはたえず賢くなっているので、この方法がうまくいかない場合があるかもしれません）。

例題 **ident**

```
$ important
Hello World
This is an extra line added later
This file is under RCS control. It's ID is
$Id: important.c,v 1.3 1998/02/11 16:55:04 rick Exp $
$ ident important
important:
     $Id: important.c,v 1.3 1998/02/11 16:55:04 rick Exp $
$
```

解説

プログラムを実行し、文字列が実行可能ファイルにも組み込まれていることを確認します。次に、**ident**コマンドを実行し、実行可能ファイルから**Id**文字列を取り出します。

RCSで実行可能ファイルに**Id**文字列を埋め込むテクニックは、顧客から問題が報告された

ファイルのバージョンを調べるときに役立ちます。RCS（およびSCCS）は、プロジェクト管理の一環として、報告された問題とその対処方法を追跡するのにも利用できます。作成したソフトウェアを販売したり配布したりする場合には、リリース間での変更点を把握することが非常に大事ですが、RCSやSCCSを使えば、この種の作業を効率よく行うことができます。

RCSについてさらに詳しく知りたい場合には、マニュアルページで`rcsintro`を参照してください。もちろん、`co`、`ci`など、RCSの各コマンドについてのマニュアルページも用意されています。

8.2.2 SCCS

SCCSは、RCSとよく似た機能を提供しています。SCCSの利点はX/Open仕様に記載があることです。このため、X/Open準拠をうたうUNIXシステムは、SCCSを必ずサポートしています。

実際的な面からいえば、RCSは非常に移植性が高く、自由に配布可能な点がメリットです。したがって、どのようなUNIXライクなシステムでも、X/Openに準拠しているかどうかに関係なく、RCSをインストールして使用することができます。ここではSCCSについての詳しい説明は省略し、RCSとのコマンドの違いを簡単に解説します。

◆ RCSとSCCSの比較

RCSとSCCSのコマンドを直接比較することは簡単にはできないので、次に示す表は単なる目安と考えてください。コマンドのオプションも両者で異なるので、SCCSを使う場合には、目的の処理を実行するのにどのオプションが必要かを調べる必要があります。

表8.1 RCSとSCCSの比較

RCS	SCCS
`rcs`	`admin`
`ci`	`delta`
`co`	`get`
`rcsdiff`	`sccsdiff`
`ident`	`what`

表に示したコマンドのほか、SCCSの`sccs`コマンドは、RCSの`rcs`コマンドと`co`コマンドにまたがる機能を持っています。たとえば、`sccs edit`と`sccs create`は、それぞれ`co -l`と`rcs -i`に相当する機能を持ちます。

8.3 マニュアルページの記述

新しいコマンドを作成する場合には、マニュアルページの作成も作業の一環として考えておく必要があります。ほとんどのマニュアルページのレイアウトは、だいたい次のようなパターンに従っています。

- ヘッダー
- NAME（名前）
- SYNOPSIS（書式）
- DESCRIPTION（説明）
- OPTIONS（オプション）
- FILES（関連ファイル）
- SEE ALSO（関連項目）
- BUGS（バグ）

これらのうち、必要のないセクションは省くことができます。Linuxのマニュアルページでは、このほかにAUTHORS（作者）というセクションもよく見かけます。

UNIXのマニュアルページは、`nroff`と呼ばれるユーティリティを使って整形されています。Linuxシステムでは、`nroff`の代わりにGNUプロジェクトの`groff`を使うことが多いようです。いずれも、古くから存在する`roff`（run-off）コマンドから発展したものです。`nroff`や`groff`に対する入力はプレーンなテキストです。一見したところ相当複雑な構文を使っているように見えますが、心配する必要はありません。UNIXでの一番簡単なプログラムの作成方法は、既存のプログラムを出発点として必要な機能を追加したり修正を加えたりする方法ですが、マニュアルページでも同じ方法を使うことができます。

本書では、`nroff`コマンドや`groff`コマンドで使うオプション、コマンド、マクロについての詳細な説明は省略し、代わりに簡単なテンプレートを示します。既存のプログラムから新しいコマンドを作成するのと同様、マニュアルページを作成するときには、このテンプレートを出発点として活用してください。

次に示すのは、この章で何度か取り上げているアプリケーション`myapp`のマニュアルページのソースです。

```
.TH MYAPP 1

.SH NAME
Myapp \- A simple demonstration application that does very little.

.SH SYNOPSIS
```

```
.B myapp
[\-option ...]

.SH DESCRIPTION
.PP
\fImyapp\fP is a complete application that does nothing useful.

.PP
It was written for demonstration purposes.

.SH OPTIONS
.PP
It doesn't have any, but lets pretend, to make this template complete:

.TP
.BI \-option
If there was an option, it would not be -option.

.SH RESOURCES
.PP
myapp uses almost no resources.

.SH DIAGNOSTICS
The program shouldn't output anything, so if you find it doing so there's
probably something wrong. The return value is zero.

.SH SEE ALSO
The only other program we know with this this little functionality is the
ubiquitous hello world application.

.SH COPYRIGHT
myapp is Copyright (c) 1996 Wrox Press.

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

.SH BUGS
There probably are some, but we don't know what they are yet.

.SH AUTHORS
Neil Matthew and Rick Stones
```

このソースからわかるように、マクロは行頭が.で始まるたいへん短いものです。一番最初の行の最後にある1は、マニュアルのセクションを表します。

このソースをコピーし、ほかのマニュアルページのソースも参考にして修正を加えれば、自分のマニュアルページを比較的簡単に作成することができます。なお、LDP（Linux Documentation Project）の一環としてJens Schweikhardtが作成したmini HOWTOのMan-Pageには、有益な情報が数多く含まれているのでぜひ参照してください。

マニュアルページのソースは、**groff**で処理します。**groff**コマンドでは、ASCIIテキスト（**-Tascii**）またはPostScript出力（**-Tps**）を生成することができます。**-man**オプションは、マニュアルページであることを示すために指定します。これで、マニュアルページ用の特別なマクロが読み込まれるようになります。

```
$ groff -Tascii -man myapp.1
```

出力は次のようになります。

```
MYAPP(1)                                                  MYAPP(1)


NAME
       Myapp  - A simple demonstration application that does very
       little.


SYNOPSIS
       myapp [-option ...]


DESCRIPTION
       myapp is a complete application that does nothing  useful.


       It was written for demonstration purposes.


OPTIONS
       It  doesn't  have any, but lets pretend, to make this tem-
       plate complete:


       -option
              If there was an option, it would not be -option.


RESOURCES
       myapp uses almost no resources.


DIAGNOSTICS
```

```
       The program shouldn't output anything, so if you  find  it
       doing  so  there's  probably  something  wrong. The return
       value is zero.


SEE ALSO
       The only other program we know with this this little func-
       tionality is the ubiquitous hello world application.


COPYRIGHT
       myapp is Copyright (c) 1996 Wrox Press.

       This  program  is  free  software; you can redistribute it
       and/or modify it under the terms of the GNU General Public
       License  as  published  by  the  Free Software Foundation;
       either version 2 of the License, or (at your  option)  any
       later version.

       This  program  is  distributed in the hope that it will be
       useful, but WITHOUT ANY WARRANTY; without even the implied
       warranty  of  MERCHANTABILITY  or FITNESS FOR A PARTICULAR
       PURPOSE.  See the GNU  General  Public  License  for  more
       details.


                                                                1



MYAPP(1)                                               MYAPP(1)


       You  should have received a copy of the GNU General Public
       License along with this program; if not, write to the Free
       Software  Foundation,  Inc.,  675  Mass Ave, Cambridge, MA
       02139, USA.


BUGS
       There probably are some, but we don't know what  they  are
       yet.


AUTHORS
       Neil Matthew and Rick Stones
```

正しく表示されることを確認したら、マニュアルページをインストールします。マニュアルページを表示する**man**コマンドは、環境変数**MANPATH**の値に従ってマニュアルページを探します。

作成したマニュアルページは、ローカルなマニュアルページ用のディレクトリに置くか、または直接`/usr/man/man1`ディレクトリに置きます。

マニュアルページを表示する必要が生じると、`man`コマンドは要求されたマニュアルページに自動的に整形を施して画面に表示します。`man`コマンドのバージョンによっては、整形済みの(圧縮された) ASCIIテキストバージョンのマニュアルページを自動的に作成、保存するものがあります。このような`man`コマンドでは、次に同じマニュアルページが要求されたときに整形済みのバージョンを利用することで処理を高速化しています。

8.4 ソフトウェアの配布

プログラムの配布に関して最も重要なことは、必要なすべてのファイルが含まれていること、そして正しいバージョンだけが含まれていることを保証することです。さいわい、インターネット上のプログミングコミュニティによって一連の優れた方法が開発されており、プログラムの配布に伴う問題はほとんど生じなくなっています。その方法をまとめると、次のようになります。

- どのUNIXマシンでも利用できる標準的なツールを使って、すべてのファイルを1つのパッケージファイルにまとめる。
- パッケージのバージョン番号を的確に管理する。
- ファイルの命名規則としてパッケージファイルにバージョン番号を含め、ユーザーがバージョンを容易に識別できるようにする。
- パッケージ内でサブディレクトリを使用し、パッケージファイルから取り出したファイルが固有のディレクトリに置かれるようになっている。このため、パッケージに含まれるファイルとそうでないファイルとをきちんと区別できる。

こうした配布方法の進化によって、プログラムは簡単かつ信頼できる形で配布できるようになっています。プログラムのインストールが容易かどうかは、プログラムそのものの性質やインストール先のシステムによって変わるのでまた別の問題ですが、少なくとも必要なファイルがすべて揃っていることはこれで保証できます。

8.4.1 patchプログラム

配布されたあとのプログラムに対して、ユーザーがバグを見つけたり、作者が機能の強化や更新を行ったりする場合があります。この場合、プログラムをバイナリで配布している作者は、しばしば単に新しいバイナリをリリースするだけですませようとします。また、ベンダーも新しい

バージョンのプログラムをリリースします。しかし、ベンダーのリビジョン管理はユーザーにとってわかりにくいことが多く、新しいバージョンでの変更内容についてもほとんど情報が得られないのが実情です。

これに対して、ソースコードでソフトウェアを配布する方法は、仕様がどのように実装されているか、また自分がどの機能を使っているかをユーザーが実際に確かめることができる点で優れています。また、プログラムの動作を隅々までチェックでき、ライセンスが許せばソースコードの一部を再利用することもできます。

ただし、問題もあります。たとえば、Linuxカーネルのソースは数MBもある大きなものですが、もし更新されたカーネルソースすべてを配布しようとすれば、相当のリソースを消費します。しかし、実際には、ソースのうち修正されている部分は全体の何パーセントかにすぎません。これはカーネルソースだけでなく、Linuxシステムを構成しているほかのパッケージなどについても同じです。

こうした問題を解決するのに役立つのが、プログラミング言語Perlの作者でもあるLarry Wallが作成した**patch**というユーティリティプログラムです。**patch**コマンドを使うと、"差分"だけを配布することができます。つまり、バージョン1のファイルと、バージョン1からバージョン2への差分ファイルがあれば、**patch**コマンドを使うことでバージョン2のファイルを作成できるのです。

実際に例を示しましょう。まず、バージョン1のファイルです。

```
This is file one
line 2
line 3
there is no line 4, this is line 5
line 6
```

次はバージョン2のファイルです。

```
This is file two
line 2
line 3
line 4
line 5
line 6
a new line 8
```

diffコマンドで差分を作成します。

```
$ diff file1.c file2.c > diffs
```

作成された**diffs**ファイルの内容は次のようになっています。

```
1c1
< This is file one
---
> This is file two
4c4,5
< there is no line 4, this is line 5
---
> line 4
> line 5
5a7
> a new line 8
```

これは実際には、あるファイルを別のファイルに変更するためのエディタコマンドです。たとえば、**file1.c**と**diffs**ファイルがあるとします。この場合、次の要領で**patch**コマンドを実行すれば、ファイルを更新することができます。

```
$ patch file1.c diffs
Hmm...  Looks like a normal diff to me...
Patching file file1.c using Plan A...
Hunk #1 succeeded at 1.
Hunk #2 succeeded at 4.
Hunk #3 succeeded at 7.
done
$
```

このコマンドを実行した結果、**file1.c**は**file2.c**と同じになります。

patchには便利な機能が用意されており、適用したパッチをもとに戻すことができます。たとえば、変更内容を破棄してもとの**file1.c**に戻したいとします。この場合には、**リバースパッチ**を意味する**-R**オプションを指定して**patch**コマンドをもう一度実行します。

```
$ patch -R file1.c diffs
Hmm...  Looks like a normal diff to me...
Patching file file1.c using Plan A...
Hunk #1 succeeded at 1.
Hunk #2 succeeded at 4.
Hunk #3 succeeded at 6.
done
$
```

これで、**file1.c**はもとの状態に戻ります。

patchコマンドにはほかにもいくつかオプションがありますが、通常は入力内容からユーザーの意図を十分にくみとり、適切な処理だけを実行してくれます。パッチに失敗した場合、**patch**コマンドは、拡張子**.rej**を持つファイルを作成します。このファイルには、パッチできなかった部分が収められています。

ソフトウェアパッチを作成する場合には、コンテキスト**diff**を生成する**-c**オプションを付け

て**diff**を実行するとよいでしょう。このコマンドを実行すると、変更箇所の前後の数行が出力に含まれるようになるので、**patch**コマンドではコンテキストが一致するかどうかを確かめてからパッチを適用することができます。また、パッチ自体も読みやすいものになります。

> **メモ** プログラムのバグを見つけて自分で修正した場合には、修正方法を説明する代わりに作者にパッチを送るとよいでしょう。そのほうが簡単ですし、情報も正確に伝わります。また、作者に対してもより親切です。

8.4.2 その他の配布ユーティリティ

UNIXのプログラムとソースは、一般にバージョン番号を含む名前と、拡張子**.tar.gz**または**.tgz**の付いたファイルで配布されます。拡張子**.tar.gz**または**.tgz**の付いたファイルは、**gzip**で圧縮されたTAR（テープアーカイブ、tape archive）ファイルです。通常の**tar**を使っている場合、**gzip**で圧縮されたTARファイルを処理するには、2つのステップを踏む必要があります。まず、TARファイルを作成してみましょう。

```
$ tar cvf myapp-1.0.tar main.c 2.c 3.c *.h myapp.1 Makefile6
main.c
2.c
3.c
a.h
b.h
c.h
myapp.1
Makefile6
$
```

TARファイルが作成されます。

```
$ ls -l *.tar
-rw-r--r--    1 rick     users       10240 Feb 11 18:30 myapp-1.0.tar
$
```

次に、圧縮プログラム**gzip**を使ってサイズを小さくします。

```
$ gzip myapp-1.0.tar
$ ls -l *.gz
-rw-r--r--    1 rick     users        1580 Feb 11 18:30 myapp-1.0.tar.gz
$
```

サイズが大幅に小さくなりました。**.tar.gz**は、単に**.tgz**という拡張子に変更することができます。

```
$ mv myapp-1.0.tar.gz myapp_v1.tgz
```

.tar.gzの代わりにピリオド1つと3文字からなる拡張子**.tgz**を使うのは、拡張子の使い方に制限のあるMS-DOSと、一部のMicrosoft Windowsソフトウェアでも問題なくファイルを利用できるようにするためです。

このファイルに格納されているファイルをもとに戻すには、圧縮を解除し、**tar**ファイルからファイルを取り出します。

```
$ mv myapp_v1.tgz myapp-1.0.tar.gz
$ gzip -d myapp-1.0.tar.gz
$ tar xvf myapp-1.0.tar
main.c
2.c
3.c
a.h
b.h
c.h
myapp.1
Makefile6
$
```


GNUバージョンの**tar**では、同じ処理をもっと簡単に実行することができます。たとえば、圧縮されたアーカイブも1つのステップで作成できます。

```
$ tar zcvf myapp_v1.tgz main.c 2.c 3.c *.h myapp.1 Makefile6
main.c
2.c
3.c
a.h
b.h
c.h
myapp.1
Makefile6
$
```

ファイルの取り出しも1つのステップで実行できます。

```
$ tar zxvf myapp_v1.tgz
main.c
2.c
3.c
a.h
b.h
c.h
myapp.1
Makefile6
$
```

ファイルを実際に取り出さずに、アーカイブにどのようなファイルが含まれているか知りたいこともあります。このような場合には、オプションを一部変更して**tar ztvf**を実行します。

◆ tarコマンドの構文

上の例では、オプションについて特に説明せずに**tar**コマンドを使いました。ここで、**tar**コマンドでよく使うオプションについて簡単にまとめておきましょう。

tarコマンドの基本的な構文は次のとおりです。

```
tar [options] [list of files]
```

ファイルリストの最初の要素はターゲットです。これまでの例ではターゲットはいずれもファイルでしたが、**tar**の名前の由来がテープアーカイブであることからわかるように、デバイスも指定することができます。ファイルリストのうちターゲットを除く要素は、アーカイブ（指定したオプションによって新しいアーカイブまたは既存のアーカイブ）に追加されます。ファイルリストにはディレクトリを含めることができます。この場合、すべてのサブディレクトリもデフォルトでアーカイブに含まれることになります。アーカイブからファイルを取り出す場合には、ディレクトリの名前を指定する必要はありません。**tar**によってファイルのフルパスが格納されているからです。

次に示すのは、これまでに使ったオプションについての簡単な説明です。

- **c**　　新しいアーカイブを作成する。
- **f**　　ターゲットはデバイスではなくファイルである。
- **t**　　アーカイブからファイルを取り出さずに内容を表示する。
- **v**　　冗長モード。処理と同時にメッセージを表示する。
- **x**　　アーカイブからファイルを取り出す。
- **z**　　GNU **tar**の中から**gzip**によるアーカイブのフィルタ処理を行う。

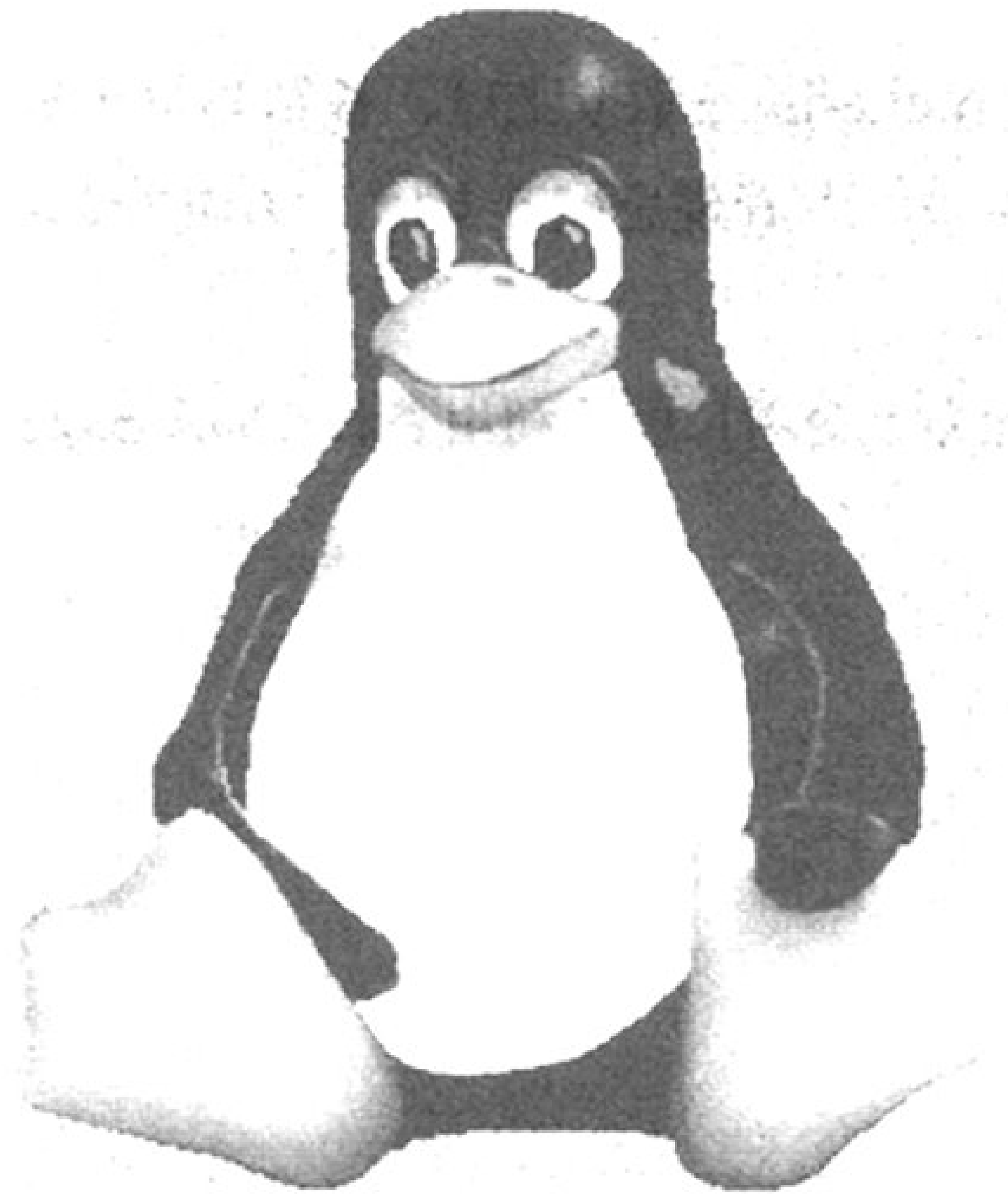
このほかにも、処理や動作の内容を細かく指定するためのオプションが数多くあります。詳細については、マニュアルページを参照してください。

8.5 この章のまとめ

この章では、プログラムの開発と配布を効率よく行うためのUNIXツールについて説明しました。

最初に`make`コマンドとmakefileを取り上げ、複数のソースファイルの管理に役立てる方法を説明しました。次にソースコード管理を取り上げ、プログラム開発の進行に合わせて変更内容を記録する方法を示しました。

最後に、プログラムの配布と更新に広く利用されている`patch`、`tar`、`gzip`を取り上げました。



第9章

デバッグ

どんなソフトウェアにも、ソースコード1000行あたり20～50の欠陥があるといわれています。人間は過ちをおかす存在であり、プログラムやライブラリもソフトウェア上のミスが原因で正しく動作しないことがあります。正しく動作しないということは、実際の動作と意図した動作とが異なることを意味しており、こうした動作をするソフトウェアシステムには**バグ**が含まれています。

このバグを取り除く作業、つまり**デバッグ**は、ソフトウェア開発に携わるプログラマから多くの時間を奪います。この章では、ソフトウェアの欠陥に注目し、間違った動作を追跡、特定するためのツールとテクニックについて説明します。デバッグは、考えられるすべての条件でプログラムの動作を検証するテストプロセスとは異なります。もちろん、デバッグとテストには互いに関連があり、テストプロセス中に見つかるバグも少なくありません。

9.1 エラーの種類

通常、バグはいくつかの基本的なことが原因で発生します。バグの検出と除去の方法は、こうした原因の種類によっても異なってきます。

◆ 仕様上のエラー

当然のことながら、プログラムの仕様が適切でなければ、作成されたプログラムも正しく動作しません。世界で最も優れたプログラマでも、間違ったプログラムを書くのは簡単です。プログラミング（または設計）に着手する前に、プログラムで実行することを明確に把握し、理解しておく必要があります。

仕様上のエラーを発見して取り除くには、プログラムに対する要求事項をよく検討し、プログラムの使い手となるユーザーとの間で要求事項について合意を得ることが大切です。

◆ 設計上のエラー

どんなサイズのプログラムでも設計は必要です。コンピュータの前にすわってキーボードから直接ソースコードを入力しただけでは、プログラムが正しく動作することはまず期待できません。プログラムの構築方法、必要なデータ構造やその使い方などについても考慮する必要があります。あらかじめ細かい点まで考えておけば、あとで手直しする手間を大幅に省くことができます。

◆ コーディングエラー

だれでも入力ミスをします。その意味では、設計図に基づくソースコードの記述は必然的に不完全な作業であり、この段階で多くのバグが混入します。バグが見つかった場合には、自分でソースコードを読み直したり、別の人間に目を通してもらったりすることも必要です。プログラム

の実装についてだれかと話をしているうちに見つかるバグも多く、こうした方法で数多くのバグを取り除くことができます。

プログラムの核となる部分を紙の上で実行してみるのもよい方法です。このプロセスは、しばしば**ドライ実行**と呼ばれます。特に重要なルーチンについては、入力値を紙に書いて1ステップずつ出力値を計算してみます。デバッグをするのに常にコンピュータが必要とは限りません。場合によっては、コンピュータを使うこと自体が問題を見えにくくすることもあります。ライブラリやコンパイラ、オペレーティングシステムを作成するような優れたプログラマでも、ミスはするものです。

9.2 デバッグの準備

この章では、コンピュータ上で使うデバッグツールとデバッグのテクニックを中心に、バグの中でも特に発生頻度の高いコーディングエラーに対処する方法を説明します。

典型的なUNIXプログラムをデバッグし、テストする場合、いくつかの独立したアプローチが考えられます。一般には、プログラムを実行し、動作するかどうかを確認します。動作しなければ、対処方法を決定する必要があります。この場合、プログラムを変更してもう一度実行することもできます（コードの検査とトライアンドエラー）。また、プログラム内部でなにが起こっているかについて詳しい情報を得る方法もあります（**インスツルメンテーション**）。あるいは、プログラムの動作を直接調べることもできます（**制御付き実行**）。一般に、デバッグには次の5つの段階があります。

- テスト：欠陥やバグの存在の検出
- 固定：バグの再現
- 特定：コード中の該当個所の特定
- 修正：コードの修正
- 検証：修正が適切であることの確認

9

9.2.1 バグのあるプログラム

まず、実際にバグのあるサンプルプログラムを用意します。この章では、このサンプルプログラムをさまざまな角度からデバッグしていきます。サンプルプログラムはある大規模なソフトウェア開発の過程で作成されたもので、`sort`という名前の関数が1つだけ含まれています。`sort`関数は、`item`型の構造体の配列に対してバブルソートアルゴリズムを実装したものです。配列の各要素は、構造体のメンバである`key`の昇順にソートされます。プログラムでは、簡単な配列に対して`sort`を呼び出します。

残念ながらもとのコードはあまり読みやすいものではなく、コメントもありません。ここでは、次のコードを出発点として使うことにします。名前は**debug1.c**とします。

```
/*   1  */  typedef struct {
/*   2  */      int   key;
/*   3  */      char *data;
/*   4  */  } item;
/*   5  */
/*   6  */  item array[] = {
/*   7  */      {3, "bill"},
/*   8  */      {4, "neil"},
/*   9  */      {2, "john"},
/*  10  */      {5, "rick"},
/*  11  */      {1, "alex"},
/*  12  */  };
/*  13  */
/*  14  */  sort(a,n)
/*  15  */  item *a;
/*  16  */  {
/*  17  */      int i = 0, j = 0;
/*  18  */      int s = 1;
/*  19  */
/*  20  */      for(; i < n && s != 0; i++) {
/*  21  */          s = 0;
/*  22  */          for(j = 0; j < n; j++) {
/*  23  */              if(a[j].key > a[j+1].key) {
/*  24  */                  item t = a[j];
/*  25  */                  a[j] = a[j+1];
/*  26  */                  a[j+1] = t;
/*  27  */                  s++;
/*  28  */              }
/*  29  */          }
/*  30  */          n--;
/*  31  */      }
/*  32  */  }
/*  33  */
/*  34  */  main()
/*  35  */  {
/*  36  */      sort(array,5);
/*  37  */  }
```

プログラムをコンパイルします。

```
$ cc -o debug1 debug1.c
```

エラーや警告は表示されず、コンパイルは正常に終了します。

プログラムを実行する前に、ソートの結果を表示するコードを追加しておきましょう。そうしないと、プログラムが動作しているかどうかを確かめることができません。ソート後に配列の値を表示する次のコードを追加し、修正後のプログラムを**debug2.c**とします。ファイルの先頭に

#include<stdio.h>を追加するので、行番号も1つずれます。

```
/*  35  */  main()
/*  36  */  {
/*  37  */          int i;
/*  38  */          sort(array,5);
/*  39  */          for(i = 0; i < 5; i++)
/*  40  */                  printf("array[%d] = {%d, %s}\n",
/*  41  */                          i, array[i].key, array[i].data);
/*  42  */  }
```

このコードはもとのプログラムの作者が記述したものではありませんが、テストのために追加しておきます。ただし、追加したコードにバグがあっては意味がないので、バグを増やさないように細心の注意が必要です。プログラムを修正したらコンパイルし、実行します。

```
$ cc -o debug2 debug2.c
$ debug2
```

結果は、プログラムを実行したUNIXシステムによって異なる可能性があります。筆者のマシンでは次のようになりました。

```
array[0] = {4, neil}
array[1] = {2, john}
array[2] = {3, bill}
array[3] = {1, alex}
array[4] = {-1, (null)}
```

別のLinuxカーネルを使っているもう1台のマシンでは、次のようになりました。

```
Segmentation fault
```

読者のUNIXマシンでも、上と類似した結果になるかもしれません。また、まったく別の結果になる可能性もあります。

いずれにしても、このコードには明らかに重大な問題があります。なんとか実行できた場合でも、配列は正しくソートされていません。また、Segmentation faultで終了した場合には、不正なメモリアクセスを行ったことを意味するシグナルがオペレーティングシステムから送られ、メモリが破壊されるのを防ぐためにプログラムは終了させられています。

不正なメモリアクセスを検出するオペレーティングシステムの機能は、ハードウェア構成と、メモリ管理の実装の詳細に依存しています。ほとんどのシステムでは、オペレーティングシステムがプログラムに割り当てるメモリは実際に使われているメモリより多いのがふつうです。このため、この領域で不正なメモリアクセスが行われた場合には、ハードウェアが不正なアクセスを検出できないことがあります。UNIXのバージョンによってはセグメンテーション違反にならないケースがあるのもそのためです。

配列へのアクセスに関する問題を特定するときには、配列の要素のサイズを大きくすると、問題を見つけやすくなることがあります。要素のサイズが増大きくなれば、それだけエラーも起こりやすくなるからです。バイト配列の終端を超えて1バイトを読み出す場合、プログラムに割り当てられたメモリはオペレーティングシステム固有の境界（8Kなど）に丸められることがあるので、場合によっては問題が顕在化しない可能性があります。

配列要素のサイズを大きくした場合、たとえばサンプルプログラムで`item`のメンバ`data`のサイズを4096文字の配列にすると、存在しない配列要素へのアクセスは、割り当てられたメモリの範囲を超える可能性が高くなります。配列の各要素は4Kなので、不正なアクセスで使用するメモリは、使用可能なメモリの終端から0〜4Kほどはみだすと考えられるからです。

実際に配列要素のサイズを次のように変更し、`debug3.c`を作成してみると、筆者の場合、以前は結果が異なった上の2つのマシンのどちらでもSegmentation faultになりました。

```
/*   4  */       char data[4096];
```

```
$ cc -o debug3 debug3.c
$ debug3
Segmentation fault
```

Linux、あるいはUNIXのシステムによってはSegmentation faultが発生しないこともあります。ANSI C標準で動作が"未定義"となっている場合、その動作に関してプログラムでは実際にどんなことでも実行できます。どうやら、サンプルプログラムもこうした非標準のCプログラムのようです。非標準のCプログラムは、非常に奇妙な動作をすることがあります。

9.2.2 コードの検査

すでに指摘したように、プログラムが期待どおりに動作しない場合には、プログラムを読み直すことが大切です。ここでは、とりあえずコードの見直しはすでに済んでいて、大きな間違いは修正されていると仮定します。

> **メモ**
> "コードの検査"は、数百行にのぼるコードを複数の開発者が共同で詳細に検討するプロセスを意味する用語としても使われます。しかし、規模は重要ではありません。いずれもコードを検査することに変わりはなく、非常に有益なテクニックである点は同じだからです。

コードの見直しを支援するツールもいくつかあります。コンパイラもそのひとつで、適切なオプションを指定することで、プログラムに構文エラーがあるかどうか調べることができます。

注意

> コンパイラの中には、変数が初期化されていない、条件で代入が使われているなど、プログラム中の疑わしい箇所に対して警告を表示するオプションを持つものがあります。たとえば、GNUコンパイラでは、次のようなオプションを指定すると、この種の警告が表示されるようになります。
>
> ```
> gcc -Wall -pedantic -ansi
> ```
>
> これですべての警告が表示されるようになり、Cの標準に準拠しているかどうかもチェックされます。問題を特定するのに役立つ有益な情報も表示されます。

少しあとでは、もうひとつのツールとして`lint`を取り上げます。`lint`はコンパイラ同様、ソースコードを解析し、不正と考えられるコードを報告します。

9.2.3 インスツルメンテーション

インスツルメンテーションとは、実行中のプログラムの動作に関する詳しい情報を収集するためにプログラムにコードを追加することです。たとえば、サンプルプログラムを紹介したときに実際に行ったように、プログラムの実行中の変数の値を表示するために`printf`を挿入する方法はよく使われます。`printf`を追加する方法は便利ですが、ソースコードを編集し、編集するたびに再コンパイルしなければならないというデメリットがあります。また、バグを解決したあと、挿入したコードを削除する必要もあります。

このような場合に活用できるインスツルメンテーションテクニックが2つあります。ひとつは、Cプリプロセッサを使ってインスツルメンテーションコードを選択的に挿入する方法です。この方法では、プログラムをコンパイルするだけで、デバッグ用のコードを含めたり除外したりすることができます。具体的には、次のようなコードを記述します。

```
#ifdef DEBUG
  printf("variable x has value = %d\n", x);
#endif
```

プログラムをコンパイルするときに、コンパイラフラグ`-DDEBUG`を指定してシンボル`DEBUG`を定義すれば、プログラムにデバッグ用コードを含めることができます。この方法をさらに進めて、次のようなデバッグ用マクロを使うこともできます。

```
#define BASIC_DEBUG 1
#define EXTRA_DEBUG 2
#define SUPER_DEBUG 4

#if (DEBUG & EXTRA_DEBUG)
  printf...
#endif
```

この方法では、常にDEBUGマクロを定義する必要がありますが、デバッグ情報のレベルを設定することが可能になります。たとえば、コンパイラフラグに-DDEBUG=5を指定すれば、BASIC_DEBUGとSUPER_DEBUGが有効になり、EXTRA_DEBUGは無効のままにしておくことができます。また、-DDEBUG=0を指定すれば、デバッグ情報は出力されません。次のような行を挿入すれば、デバッグが必要ない場合にコマンドラインでDEBUGを指定する必要もなくなります。

```
#ifndef DEBUG
#define DEBUG 0
#endif
```

次に示すのは、デバッグ情報を出力するときに役立つCプリプロセッサのマクロです。これらのマクロは、コンパイルに関する情報を表す値に展開されます。

表9.1 デバッグに役立つマクロ

マクロ	説明
__LINE__	現在の行番号を表す10進定数
__FILE__	現在のファイル名を表す文字列
__DATE__	"Mmm dd yyyy"形式で現在の日付を表す文字列
__TIME__	"hh:mm:ss"形式で現在の時刻を表す文字列

シンボルの前後にアンダスコアが2つずつ付いている点に注意してください。これは、標準のプリプロセッサシンボルとして一般的であり、これらのシンボルと自分で定義するシンボルとが衝突しないように注意する必要があります。なお、上の表で使われている"現在"とは、プリプロセッシングが実行された時点、つまりコンパイラが実行されてファイルが処理された日付と時刻を意味します。

例題 デバッグ情報

次のcinfo.cでは、デバッグが有効になっている場合、コンパイルの日付と時刻に関する情報を表示します。

```
#include <stdio.h>

int main()
{
#ifdef DEBUG
    printf("Compiled: " __DATE__ " at " __TIME__ "\n");
    printf("This is line %d of file %s\n", __LINE__, __FILE__);
#endif
    printf("hello world\n");
    exit(0);
}
```

-DDEBUGを指定し、デバッグを有効にしてプログラムをコンパイルすると、コンパイル情報が表示されます。

```
$ cc -o cinfo -DDEBUG cinfo.c
$ cinfo
Compiled: Feb  4 1998 at 14:01:08
This is line 7 of file cinfo.c
hello world
$
```

解説

コンパイラから呼び出されたCプリプロセッサは、コンパイル中のファイルの現在の行とファイル名を記録しています。Cプリプロセッサは、__LINE__および__FILE__というシンボルを見つけると、現在の（コンパイル時の）値に置き換えます。コンパイルの日付と時刻についても同様に処理されます。

__DATE__と__TIME__は文字列なので、これらをprintfの書式指定文字列と連結して使っている点に注意してください。ANSI Cでは、隣接する複数の文字列は1つの文字列として扱うと規定しています。

◆ 再コンパイルの不要なデバッグ

#ifdef DEBUGを使ってprintfを挿入する場合は、プログラムの再コンパイルが必要になります。ここでは、もうひとつの方法として、#ifdef DEBUGを使わずにprintfでデバッグ情報を表示する方法を示します。

その方法とは、グローバル変数をデバッグフラグとして用意し、コマンドラインオプションの-dでこの値を指定することにより、プログラムのリリース後にもデバッグの有効と無効を切り替えることができるようにする方法です。プログラムの要所要所に次のようなコードを挿入しておきます。

```
if (debug) {
sprintf(msg, ...)
write_debug(msg)
}
```

通常、デバッグ出力はstderrに書き込むようにします。プログラムの性質上、stderrを使うことが難しい場合には、syslog関数が提供するログ機能を使う方法もあります。

問題を解決するために開発時に上のようなコードを挿入した場合には、問題が解決したあともそのままにしておくとよいでしょう。これらのコードは、プログラムのリリース後にも役立つからです。たとえば、ユーザーが問題を見つけたとします。この場合、デバッグを有効にしてプログラムを実行すれば、ユーザー自身でエラーの診断を行うことができます。また、ユーザーから

9

報告を受ける作者の側も、Segmentation faultのようなあいまいなメッセージではなく、問題が発生したときにプログラムがどのような処理を行っていたかを正確に把握することができます。もしデバッグ情報を利用できなければ、ユーザーは自分が行った操作を伝えるしかありませんが、ユーザーの操作と実際のプログラムの実行内容とでは、情報としての価値に大きな違いがあります。

グローバル変数を使う方法には欠点もあります。それは、プログラムのサイズが増えることです。ただし、ほとんどの場合、これは単に表面的な問題にすぎません。プログラムのサイズが2〜3割大きくなっても、パフォーマンスが影響を受けることはまずありません。現実にパフォーマンスに影響が出るのは相当なサイズの違いがある場合であり、2倍程度ではそれほど差はありません。

9.2.4 制御付き実行

さて、サンプルプログラムに戻りましょう。このプログラムにはバグがあります。プログラム実行中の変数の値を表示するコードを追加する方法についてはすでに説明しました。もうひとつの方法として、デバッガを使ってプログラムの実行と同時にプログラムの動作や状態を逐一確認する方法があります。

UNIXシステムで利用できるデバッガにはさまざまなものがあります。どのようなデバッガが使用できるかは、UNIXシステムのベンダーによっても異なります。よく使われるものとしては、**adb**、**sdb**、**dbx**などがあります。高機能なデバッガでは、ソースコードレベルでプログラムの状態を詳細に調べることも可能です。**sdb**や**dbx**、Linuxで使用できるGNUデバッガの**gdb**はそうしたデバッガの例です。**gdb**には、操作性を向上するためのフロントエンドもあります。代表的なフロントエンドとしては、**xxgdb**と**tgb**があります。

デバッガでプログラムをデバッグするには、いくつかの特別なコンパイラオプションを指定してプログラムをコンパイルする必要があります。これらのオプションは、コンパイラに対して追加のデバッグ情報をプログラムに埋め込むように指示するためのものです。デバッグ情報にはシンボルと行番号が含まれており、デバッガはこれらの情報を使ってプログラムとそのソースコードとを対応付けます。

デバッグ用にプログラムをコンパイルするときには、**-g**フラグを指定します。このフラグは、デバッグの対象となる各ソースコードのコンパイル時に指定する必要があります。

デバッグ情報を含めると、作成される実行可能ファイルのサイズは数倍から最大で10倍程度にふくれあがります。デバッグを終えたら、デバッグ情報を削除してからプログラムをリリースするとよいでしょう。

9.3 gdbによるデバッグ

ここでは、GNUデバッガ**gdb**を使ってプログラムをデバッグします。フリーで配布されている**gdb**は、多くのUNIXプラットフォームで使用でき、Linuxシステムではデフォルトのデバッガになっています。**gdb**は多種多様なプラットフォームに移植されており、組み込みリアルタイムシステムのデバッグに使用することもできます。

9.3.1 gdbの起動

さっそくサンプルプログラムをデバッグ用にコンパイルし、**gdb**を起動してみましょう。

```
$ cc -g -o debug3 debug3.c
$ gdb debug3
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.16 (i486-unknown-linux),
Copyright 1996 Free Software Foundation, Inc...
(gdb)
```

gdbには充実したオンラインヘルプがあります。また、**info**プログラムや**emacs**から表示できる**info**形式の詳細なマニュアルも用意されています。

```
(gdb) help
List of classes of commands:

running -- Running the program
stack -- Examining the stack
data -- Examining data
breakpoints -- Making program stop at certain points
files -- Specifying and examining files
status -- Status inquiries
support -- Support facilities
user-defined -- User-defined commands
aliases -- Aliases of other commands
obscure -- Obscure features
internals -- Maintenance commands

Type "help" followed by a class name for a list of commands in that class.
Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
(gdb)
```

9.3.2 プログラムの実行

プログラムを実行するには**run**コマンドを使います。**run**コマンドに引数を指定すると、デバッグするプログラムに引数として渡されます。サンプルプログラムの場合には、引数を指定する必要はありません。

ここでは、サンプルプログラムを実行したときにSegmentation faultが発生することを前提に説明を進めます。実際にSegmentation faultが発生しない場合には、以下の画面出力を見ながら解説を読み進んでください。メモリアクセスの問題を修正した**debug4.c**からは、デバッグの手順を実際に画面で確認できるようになります。

```
(gdb) run
Starting program: /usr/neil/debug3

Program received signal SIGSEGV, Segmentation fault.
0x8048603 in sort (a=0x8049764, n=5) at debug3.c:26
26   /*  26  */                              a[j] = a[j+1];
(gdb)
```

以前と同様、プログラムは正しく動作しません。**gdb**はプログラムの実行に失敗した時点で、その理由と場所を表示します。これを出発点に問題の原因を調べることになります。

9.3.3 スタックトレース

プログラムは、ソースファイル**debug3.c**の26行目にあたる**sort**関数の途中で止まりました。もし追加のデバッグ情報（**cc -g**）を含めずにプログラムをコンパイルしていたとすると、プログラムがどこで失敗したかわかりません。また、変数名を使ってデータを調べることもできません。

プログラムがどのような道筋をたどって中断されるに至ったかは、**backtrace**コマンドで調べることができます。

```
(gdb) backtrace
#0  0x8048603 in sort (a=0x8049764, n=5) at debug3.c:26
#1  0x8048682 in main () at debug3.c:38
#2  0x804842e in ___crt_dummy__ ()
(gdb)
```

サンプルプログラムはごく単純なプログラムです。ほかの関数から別の関数をいくつも呼び出すといった複雑なことはしていないので、トレースの結果もたいへん短くなっています。上の出力からは、同じファイル**debug3.c**の38行目の**main**関数から**sort**関数が呼び出されていることがわかります。一般のプログラムはサンプルプログラムよりはるかに複雑であることが多く、**backtrace**を使ってエラーが発生するまでの経路を調べることになります。このコマンドは、プログラム内のさまざまな場所から呼び出される関数をデバッグするときに特に便利です。

backtraceコマンドは、短くbtと指定することもできます。また、ほかのデバッガとの互換性を確保する目的から、同じ機能を持つwhereコマンドも用意されています。

9.3.4 変数の値の表示

プログラムが中断されたときにgdbによって表示された情報とスタックフレームの情報には、関数の引数の値が含まれています。

それによると、sort関数が呼び出されたときのパラメータaの値は、0x8049764です。これは配列のアドレスを表しています。この値は、使用しているシステムやコンパイラ、オペレーティングシステムによって異なります。

問題の26行目では、配列の1つの要素を別の要素に代入しています。

```
26   /*  26  */                                     a[j] = a[j+1];
```

デバッガを使うと、関数のパラメータ、ローカル変数、グローバルデータの内容を表示することができます。変数やその他の式の値を表示するには、printコマンドを使います。

```
(gdb) print j
$1 = 4
```


ローカル変数jの値は4です。つまり、プログラムでは次のステートメントを実行しようとしたことになります。

```
a[4] = a[4+1];
```

sort関数に渡された配列arrayの要素は5つしかないので、配列の添字は0から4までということになります。つまり、上のステートメントでは、存在しない要素array[5]から値を読み取っています。これは、ループ変数jの値が不正であることが原因です。

printコマンドを使えば、関数に渡された配列の要素の内容を表示することもできます。gdbでは、変数の値や配列の要素、ポインタを表示するために、Cで有効な式のほとんどを使用することができます。

```
(gdb) print a[3]
$2 = {key = 1, data = "alex", '\000' <repeats 4091 times>}
(gdb)
```

gdbは、コマンドの結果を疑似変数$<number>に保持しています。また、最後の結果は常に$に、またその1つ前の結果は$$に格納されます。このことを利用すれば、ある結果をほかのコマンドの中で使うことができます。

```
(gdb) print j
$3 = 4
(gdb) print a[$-1].key
$4 = 1
(gdb)
```

9.3.5 プログラムリストの表示

gdbの中からプログラムのソースコードを表示するには、listコマンドを使います。listコマンドを実行すると、ソースコードのうち、現在位置の前後の部分が表示されます。続けてコマンドを実行すると、それ以降の部分が表示されます。

```
(gdb) list
21   /*  21  */        for(; i < n && s != 0; i++) {
22   /*  22  */                s = 0;
23   /*  23  */                for(j = 0; j < n; j++) {
24   /*  24  */                        if(a[j].key > a[j+1].key) {
25   /*  25  */                                item t = a[j];
26   /*  26  */                                a[j] = a[j+1];
27   /*  27  */                                a[j+1] = t;
28   /*  28  */                                s++;
29   /*  29  */                        }
30   /*  30  */                }
(gdb)
```

23行目には、変数jがnより小さいときに実行されるループがあります。サンプルプログラムではnは5なので、jの値は最終的に4になります。しかしこれでは、a[4]とa[5]を比較し、その結果によってこれら2つの要素を入れ替えるという操作を行うことになります。この問題は、とりあえずループの終了判定条件をj < n-1とすることで回避できます。

実際にプログラムを変更して結果を確認してみましょう。変更箇所は次に示すとおりです。修正後のプログラムはdebug4.cとします。

```
/*  23  */          for(j = 0; j < n-1; j++) {
```

```
$ cc -g -o debug4 debug4.c
$ debug4
array[0] = {2, john}
array[1] = {1, alex}
array[2] = {3, bill}
array[3] = {4, neil}
array[4] = {5, rick}
```

残念ながらまだ正しく動作しないようです。ソート後の結果が正しい順序になっていません。今度は、gdbを使ってプログラムを1ステップずつ実行してみましょう。

9.3.6 ブレークポイントの設定

プログラムのどこで問題が発生しているかを特定するには、実行中のプログラムの動作を確かめる必要があります。このような場合に役立つのが**ブレークポイント**です。ブレークポイントを設定すると、任意の時点でプログラムを停止させることができます。ブレークポイントに達すると、プログラムの実行は中断され、デバッガに制御が戻されるので、変数の値を表示したりすることができます。必要な操作を行ったら、プログラムの実行を再開することができます。

`sort`関数には2つのループがあります。ループ変数`i`を使った外側のループは、配列中の各要素に対して一度だけ実行されます。内側のループでは、現在注目している要素と、この要素のキーよりも小さいキーを持つ要素とを入れ替えます。これで、ちょうど水中から水面に向かって空気の泡（バブル）が上っていくように、より小さな要素が配列の前に移動します。また、外側のループが実行されるたびに、要素中の最大のものが配列の最後に向かって移動するはずです。では、外側のループでプログラムを停止して配列の状態を実際に確かめてみましょう。

ブレークポイントの設定に関係するコマンドはたくさんあります。どのようなコマンドがあるかは、`help breakpoint`で表示することができます。

```
(gdb) help breakpoint
Making program stop at certain points.

List of commands:

awatch -- Set a watchpoint for an expression
rwatch -- Set a read watchpoint for an expression
watch -- Set a watchpoint for an expression
catch -- Set breakpoints to catch exceptions that are raised
break -- Set breakpoint at specified line or function
clear -- Clear breakpoint at specified line or function
delete -- Delete some breakpoints or auto-display expressions
disable -- Disable some breakpoints
enable -- Enable some breakpoints
thbreak -- Set a temporary hardware assisted breakpoint
hbreak -- Set a hardware assisted  breakpoint
tbreak -- Set a temporary breakpoint
condition -- Specify breakpoint number N to break only if COND is true
commands -- Set commands to be executed when a breakpoint is hit
ignore -- Set ignore-count of breakpoint number N to COUNT

Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
```

ここでは、21行目にブレークポイントを設定してプログラムを実行します。

```
$ gdb debug4
(gdb) break 21
Breakpoint 1 at 0x80484b1: file debug4.c, line 21.
```

```
(gdb) run
Starting program:
/home/kasai/project/cosmo/blp/translat/tmp/chap09/reviewed/debug4

Breakpoint 1, sort (a=0x8049764, n=5) at debug4.c:21
21    /*  21  */      for(; i < n && s != 0; i++) {
```

配列の値を表示したあと、**cont**コマンドを使えばプログラムの実行を再開することができます。この場合、プログラムは次のブレークポイントまで実行されます。現在の例では、21行目がもう一度実行されるまでプログラムが実行されることになります。同時に複数のブレークポイントをアクティブにすることもできます。

```
(gdb) print array[0]
$1 = {key = 3, data = "bill", '\000' <repeats 4091 times>}
```

いくつかの要素を続けて表示するには、指定した数の配列要素を表示させる**@<number>**という構文を使います。たとえば、配列**array**のすべての要素を表示するには、次のようにします。

```
(gdb) print array[0]@5
$2 = {{key = 3, data = "bill", '\000' <repeats 4091 times>},
        {key = 4, data = "neil", '\000' <repeats 4091 times>},
        {key = 2, data = "john", '\000' <repeats 4091 times>},
        {key = 5, data = "rick", '\000' <repeats 4091 times>},
        {key = 1, data = "alex", '\000' <repeats 4091 times>}}
```

gdbからの出力は、読みやすくするために若干修正しています。ループを初めて実行する時点では、配列の内容は変化していません。プログラムの実行を再開すると、配列**array**の内容がプログラムの実行に伴って徐々に変化するのがわかります。

```
(gdb) cont
Continuing.

Breakpoint 1, sort (a=0x8049764, n=4) at debug4.c:21
21    /*  21  */      for(; i < n && s != 0; i++) {
(gdb) print array[0]@5
$3 = {{key = 3, data = "bill", '\000' <repeats 4091 times>},
       {key = 2, data = "john", '\000' <repeats 4091 times>},
       {key = 4, data = "neil", '\000' <repeats 4091 times>},
       {key = 1, data = "alex", '\000' <repeats 4091 times>},
       {key = 5, data = "rick", '\000' <repeats 4091 times>}}
```

displayコマンドを使うと、プログラムがブレークポイントで停止するたびに自動的に配列の内容を表示させることができます。

```
(gdb) display array[0]@5
1: array[0] @ 5 = {{key = 3, data = "bill", '\000' <repeats 4091 times>},
```

```
        {key = 2, data = "john", '\000' <repeats 4091 times>},
        {key = 4, data = "neil", '\000' <repeats 4091 times>},
        {key = 1, data = "alex", '\000' <repeats 4091 times>},
        {key = 5, data = "rick", '\000' <repeats 4091 times>}}
```

また、ブレークポイントに達したときの動作を変更し、プログラムを停止させる代わりに、指定したデータを表示してプログラムの実行はそのまま継続することもできます。この場合には、**commands**コマンドを使います。**commands**コマンドでは、ブレークポイントで実行するデバッガコマンドを指定することができます。現在の例では、すでに**display**コマンドを設定しているので、あとは実行を継続するコマンドを設定するだけです。

```
(gdb) commands
Type commands for when breakpoint 1 is hit, one per line.
End with a line saying just "end".
>cont
>end
```

プログラムの実行を再開すると、今度は最後まで実行され、外側のループを回るたびに配列の値が表示されます。

```
(gdb) cont
Continuing.

Breakpoint 1, sort (a=0x8049764, n=3) at debug4.c:21
21    /*  21  */      for(; i < n && s != 0; i++) {
1: array[0] @ 5 = {{key = 2, data = "john", '\000' <repeats 4091 times>},
        {key = 3, data = "bill", '\000' <repeats 4091 times>},
        {key = 1, data = "alex", '\000' <repeats 4091 times>},
        {key = 4, data = "neil", '\000' <repeats 4091 times>},
        {key = 5, data = "rick", '\000' <repeats 4091 times>}}

Breakpoint 1, sort (a=0x8049764, n=2) at debug4.c:21
21    /*  21  */      for(; i < n && s != 0; i++) {
1: array[0] @ 5 = {{key = 2, data = "john", '\000' <repeats 4091 times>},
        {key = 1, data = "alex", '\000' <repeats 4091 times>},
        {key = 3, data = "bill", '\000' <repeats 4091 times>},
        {key = 4, data = "neil", '\000' <repeats 4091 times>},
        {key = 5, data = "rick", '\000' <repeats 4091 times>}}
array[0] = {2, john}
array[1] = {1, alex}
array[2] = {3, bill}
array[3] = {4, neil}
array[4] = {5, rick}

Program exited with code 025.
(gdb)
```

出力の最後では、プログラムが異常な終了コードで終了しています。これは、プログラムの中

でexitを呼び出しておらず、mainも値を返していないためです。したがって、サンプルプログラムの場合、終了コードには意味がありません。実際に意味のある終了コードを返すには、exitを呼び出す必要があります。

プログラムのデバッグに戻りましょう。外側のループは期待した回数実行されていないようです。デバッガの出力からわかるように、ループの終了条件で使われているパラメータnの値は、ブレークポイントを通るたびに減っています。ループが十分な回数実行されないのは、このことが原因と考えられます。問題は、31行目のnをデクリメントしている部分にあります。

```
/*  31  */                n--;
```

この行は、外側のループを回った時点で配列arrayの最大の要素が一番最後にくることを利用し、ソートの対象となる要素を減らしてプログラムの最適化を意図したものです。しかし、これが外側のループと干渉し、問題の原因を作っています。解決方法はいくつかありますが、そのうち最も簡単なのはこの行を削除してしまうことです。さっそくデバッガを使ってパッチを適用し、この変更によって問題が解決されるかどうか確かめてみましょう。

9.3.7 デバッガによるパッチの適用

デバッガを使ってブレークポイントを設定し、変数の値を表示する方法についてはすでに見ましたが、ブレークポイントでなんらかの処理を行うようにすると、ソースコードの変更と再コンパイルを行うことなく、**パッチ**を適用することができます。現在の例では、31行目でプログラムを停止し、変数nの値をインクリメントします。これで、31行目が実行されてもnの値は変わらないことになります。

プログラムを最初から実行し直します。まず、これまでに設定したブレークポイントと変数の自動表示を削除します。有効になっているブレークポイントと自動表示を調べるには、infoコマンドを使います。

```
(gdb) info display
Auto-display expressions now in effect:
Num Enb Expression
1:   y  array[0] @ 5
(gdb) info break
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x080484b1 in sort at debug4.c:21
        breakpoint already hit 4 times
        cont
```

ブレークポイントや自動表示は無効にすることも、完全に削除してしまうこともできます。無効にした場合には、必要に応じてあとでもう一度有効にすることができます。

```
(gdb) disable break 1
(gdb) disable display 1
(gdb) break 31
Breakpoint 2 at 0x8048650: file debug4.c, line 31.
(gdb) commands 2
Type commands for when breakpoint 2 is hit, one per line.
End with a line saying just "end".
>set variable n = n+1
>cont
>end
(gdb) run
Starting program:
/home/kasai/project/cosmo/blp/translat/tmp/chap09/reviewed/debug4

Breakpoint 2, sort (a=0x8049764, n=5) at debug4.c:31
31   /*  31  */              n--;

Breakpoint 2, sort (a=0x8049764, n=5) at debug4.c:31
31   /*  31  */              n--;

Breakpoint 2, sort (a=0x8049764, n=5) at debug4.c:31
31   /*  31  */              n--;

Breakpoint 2, sort (a=0x8049764, n=5) at debug4.c:31
31   /*  31  */              n--;

Breakpoint 2, sort (a=0x8049764, n=5) at debug4.c:31
31   /*  31  */              n--;
array[0] = {1, alex}
array[1] = {2, john}
array[2] = {3, bill}
array[3] = {4, neil}
array[4] = {5, rick}

Program exited with code 025.
(gdb)
```

プログラムは最後まで実行され、今度は正しい結果が表示されました。これで、ソースファイルを変更し、処理するデータの量を増やしてさらにテストを続けることができます。

9.3.8 gdbのその他の機能

GNUデバッガ**gdb**は非常に強力なツールです。**gdb**を使うと、実行中のプログラムの内部状態について非常に多くの情報を得ることができます。ハードウェアブレークポイントをサポートしているシステムでは、**gdb**を使って変数の変化をリアルタイムで監視することもできます。ハードウェアブレークポイントをサポートしていないシステムでも、式をウォッチ（監視）し、プログラムの中のどの部分で式の計算が行われたかにかかわらず、式が特定の値になったときにプログラムを停止させることができます。

ブレークポイントは、到達回数や条件を指定して設定することができます。この場合、指定した回数に達したとき、または条件が成立したときにのみプログラムを停止させることができます。

gdbは、すでに実行されているプログラムに対して自分自身をアタッチすることができます。この機能は、クライアント／サーバーシステムをデバッグするときに非常に便利です。たとえば、サーバープロセスの停止と再開を行わずに実行中のサーバープロセスをデバッグすることができます。また、最適化とデバッグ情報の両方を有効にするために、`gcc -O -g`でプログラムをコンパイルすることもできます。ただし、最適化を行うとコードの順序が入れ替わることがあり、ステップ実行を行うとあちこちに飛ぶ可能性があります。このため、オリジナルのソースコードの処理の流れを追いにくくなるかもしれません。

gdbはGNU General Public Licenseの条件のもとで配布されており、ほとんどのUNIXシステムで使用することができます。ぜひgdbの使い方に習熟してプログラムのデバッグに役立ててください。

9.4 その他のデバッグツール

UNIXシステムでは、gdbなどの純然たるデバッガのほかにも、デバッグを支援するさまざまなツールを利用することができます。これらのツールの中には、プログラムに関する静的な情報を提供するものと、動的な分析を可能にするものとがあります。

静的な分析では、プログラムのソースコードだけに対象を絞った情報が得られます。たとえば、ctags、cxref、cflowといったプログラムは、ソースファイルを処理し、関数呼び出しやその位置に関して有益な情報を表示します。

動的な分析では、実行中のプログラムの動作に関する情報を得ることができます。たとえば、profやgprofなどのプログラムは、実行された関数とその時間に関する情報を表示します。

ここでは、これらのツールについて簡単に取り上げます。ツールの中にはシステムによって利用できないものもありますが、ほとんどのツールにはフリーで配布されているバージョンがあります。

9.4.1 プログラムのミスを指摘するlint

オリジナルのUNIXシステムには、lintと呼ばれるユーティリティがありました。このlintは基本的にはCコンパイラのフロントエンドと考えることができ、“常識的な”観点からソースコードを検査して警告を表示する機能を備えていました。たとえば、値が設定されずに使われている変数や、関数の引数が使われていない箇所などを検出することができました。

今日一般に使われている新しいCコンパイラでは、（コンパイル時のパフォーマンスに多少影響するものの）同種の警告を表示することができます。lint自体は、Cの標準化によって存在意

義が薄れています。lintは初期のCコンパイラに基づいて作成されたツールなので、ANSI構文を適切に処理することができません。UNIXで利用可能なものとしては、商用バージョンのlintがいくつか存在します。Linuxでは、形式仕様用ツールの作成を目的としたMITのプロジェクトLarchの一環として提供されているlclintを利用できます。lclintは、ソースコードをチェックして有益なコメントを表示してくれます。

次に示すのは、サンプルプログラムの最初期のバージョンに対してlclintを実行したときのようすです。

```
$ lclint debug0.c
LCLint 2.3i --- 15 Sep 97

debug0.c: (in function sort)
debug0.c:20,32: Variable s used before definition
  An rvalue is used that may not be initialized to a value on some execution
  path. (-usedef will suppress message)
debug0.c:32,14: Path with no return in function declared to return int
  There is a path through a function declared to return a value on which there
  is no return statement. This means the execution may fall through without
  returning a meaningful result to the caller. (-noret will suppress message)
debug0.c: (in function main)
debug0.c:36,17: Return value (type int) ignored: sort(array, 5)
  Result returned by function call is not used. If this is intended, can cast
  result to (void) to eliminate message. (-retvalint will suppress message)
debug0.c:37,14: Path with no return in function declared to return int
debug0.c:6,18: Variable exported but not used outside debug0: array
  A declaration is exported, but not used outside this module. Declaration can
  use static qualifier. (-exportlocal will suppress message)
debug0.c:14,13: Function exported but not used outside debug0: sort
   debug0.c:15,17: Definition of sort

Finished LCLint checking --- 6 code errors found
$
```

初期化されていない変数が使われている、関数の戻り値の型と実際に関数が返す値が合っていない、ローカル変数やローカル関数がstaticとして宣言されていない、といったエラーが報告されています。このうち、最初のもの以外は実際のプログラムの動作には影響しません。

最初のエラーは問題です。debug0.cの該当部分は次のようになっていました。

```
/*  18  */        int s;
/*  19  */
/*  20  */        for(; i < n && s != 0; i++) {
/*  21  */                s = 0;
```

たしかに、変数sは初期化されずに使われています。

こうしたエラーは、実際にデバッグを行わなくてもコードをチェックすれば見つけることができます。

9.4.2 関数呼び出しのチェック

ctags、**cxref**、**cflow**の3つのユーティリティはX/Open仕様の一部となっています。したがって、ソフトウェア開発機能を備え、UNIXブランドを掲げるシステムでは必ず存在していなければなりません。

◆ ctags

ctagsプログラムは関数のインデックスを作成し、ちょうど本の索引（インデックス）のように、関数が使われている場所についての情報を出力します。

```
ctags [-a] [-f filename] sourcefile sourcefile ...
ctags -x sourcefile sourcefile ...
```

デフォルトでは、**ctags**は入力ソースファイルのいずれかで宣言されている各関数について、次の形式の行を含む**tags**という名前のファイルを現在のディレクトリに作成します。

```
announce        app_ui.c        /^static void announce(void) $/
```

ファイルの各行は、関数名、関数が宣言されているファイル、ファイル内で関数を検索するときに使用できる正規表現から構成されています。

オプション**-x**を指定して**ctags**を実行した場合には、次の形式の行が標準出力に出力されます。

```
find_cat        403 app_ui.c        static cdc_entry find_cat(
```

オプション**-f filename**を指定すると、**ctags**からの出力を別のファイルにリダイレクトすることができます。また、オプション**-a**を指定すれば、出力を既存のファイルに追加することができます。

◆ cxref

cxrefプログラムは、Cソースコードを分析してクロスリファレンスを生成します。具体的には、各シンボル（変数、**#define**、関数）がプログラムのどこに出現しているかを表示します。

cxrefにはいろいろなバージョンがあるので、正確な構文と使い方についてはシステムのマニュアルや**man**ページ、プログラムに付属するドキュメントなどを参照してください。

◆ cflow

cflowプログラムは、どの関数がどの関数を呼び出しているか、どの関数がどの関数から呼び出されているかなどを示す**関数呼び出しツリー**を表示します。**cflow**を使うとプログラムの構造

を把握しやすくなるので、プログラムの動作や、関数を変更したときの影響などを調べるときに役立ちます。cflowのいくつかのバージョンでは、ソースコードだけでなくオブジェクトファイルを処理できるものもあります。詳細については、マニュアルページを参照してください。

次に示すのは、インターネットから入手可能で、Marty Leisnerがメンテナンスしているcflow (cflow-2.0) の出力例です。

```
1    file_ungetc {prcc.c 997}
2    main {prcc.c 70}
3        getopt {}
4        show_all_lists {prcc.c 1070}
5            display_list {prcc.c 1056}
6                printf {}
7            exit {}
8        exit {}
9        usage {prcc.c 59}
10           fprintf {}
11           exit {}
```

この出力例では、mainがshow_all_listsを呼び出し、show_all_listsはdisplay_listを、またdisplay_listはprintfを呼び出していることがわかります。

このバージョンのcflowには、逆の流れ図を出力する-iオプションがあります。-iオプションを指定すると、ある関数をどの関数が呼び出しているかがわかります。実際に例を示しましょう。

```
19   display_list {prcc.c 1056}
20       show_all_lists {prcc.c 1070}
21   exit {}
22       main {prcc.c 70}
23       show_all_lists {prcc.c 1070}
24       usage {prcc.c 59}
...
74   printf {}
75       display_list {prcc.c 1056}
76       maketag {prcc.c 487}
77   show_all_lists {prcc.c 1070}
78       main {prcc.c 70}
...
99   usage {prcc.c 59}
100      main {prcc.c 70}
```

この出力例では、exit関数がmain、show_all_lists、usageの各関数から呼び出されていることがわかります。

9

9.4.3 実行プロファイル

プログラムのパフォーマンス上の問題を特定するときに便利なテクニックとして、**実行プロファイル**があります。特別なコンパイラオプションを指定し、補助プログラムを使ってプログラムのプロファイルを表示すると、プログラムのどの部分の実行にどれだけの時間が費やされたかを知ることができます。

◆ prof／gprof

profプログラム（またはGNUの**gprof**）は、プログラムを実行したときに生成される実行トレースファイルに基づいて、プログラム各部の実行時間や実行回数についての情報を表示します。プロファイルを取得するには、次のように**-pg**フラグを指定してプログラムをコンパイルします。

```
$ cc -pg -o program program.c
```

この方法でプログラムをコンパイルすると、特別なバージョンのCライブラリとリンクされ、監視用のコードが埋め込まれます。プログラムの監視方法はシステムによって異なりますが、一般に、プログラムの実行を繰り返し中断し、実行された箇所を記録するという方法で実現されています。プログラムの監視によって得られたデータは、現在のディレクトリの**mon.out**（**gprof**の場合には**gmon.out**）というファイルに書き込まれます。

```
$ program
$ ls -ls
   2 -rw-r--r--   1 neil     users        1294 Feb  4 11:48 gmon.out
```

profまたは**gprof**プログラムは、この監視データを読み取ってレポートを作成します。プログラムのオプションについては、マニュアルページを参照してください。次に示すのは**gprof**からの出力の一部です。

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 24.53      0.13     0.13                             mcount
 15.09      0.21     0.08    17047     4.69     9.97  print_long_format
 13.21      0.28     0.07    17533     3.99     3.99  attach
 13.21      0.35     0.07    17050     4.11     8.10  gobble_file
  5.66      0.38     0.03    17097     1.75     1.75  quote_filename
  5.66      0.41     0.03      489    61.35    65.32  extract_dirs_from_files
  3.77      0.43     0.02      489    40.90   776.91  print_dir
  3.77      0.45     0.02                             compare_name
  1.89      0.46     0.01    18023     0.55     0.55  file_interesting
  1.89      0.47     0.01    17097     0.58     2.92  print_fancy_name
  1.89      0.48     0.01    17097     0.58     2.34  print_name_with_quoting
```

```
1.89      0.49     0.01    17047     0.59      0.59    ftypelet
1.89      0.50     0.01    17047     0.59      0.59    getgroup
1.89      0.51     0.01    17047     0.59      1.76    mode_string
1.89      0.52     0.01    17047     0.59      0.59    setst
1.89      0.53     0.01      488    20.49    368.85    print_current_files
0.00      0.53     0.00    51141     0.00      0.00    rwx
0.00      0.53     0.00    18619     0.00      0.00    xmalloc
0.00      0.53     0.00    17591     0.00      0.00    xstrdup
0.00      0.53     0.00    17050     0.00      0.00    lstat
0.00      0.53     0.00    17050     0.00      0.00    safe_lstat
0.00      0.53     0.00    17047     0.00      0.00    getuser
0.00      0.53     0.00      489     0.00      0.00    clear_files
0.00      0.53     0.00      489     0.00      0.00    queue_directory
0.00      0.53     0.00      489     0.00      0.00    sort_files
0.00      0.53     0.00      486     0.00      0.00    is_not_dot_or_dotdot
0.00      0.53     0.00       50     0.00      0.00    get_link_name
0.00      0.53     0.00       50     0.00      0.00    make_link_path
0.00      0.53     0.00        2     0.00      0.00    xrealloc
0.00      0.53     0.00        1     0.00      0.00    decode_switches
0.00      0.53     0.00        1     0.00      0.00    error
0.00      0.53     0.00        1     0.00 380000.00    main
```

9.5 アサーション

プログラムの開発中に条件コンパイルによって**printf**などのデバッグコードを挿入する方法は一般的ですが、プログラムをリリースする際の現実的な問題として、デバッグ用のコードをそのまま残すことができない場合もあります。しかし、リリース後のプログラムの実行中に"起きるはずのない問題が起きる"、つまりコーディングエラーというより、誤った仮定が原因で問題が発生することも少なくありません。たとえば、入力パラメータが一定の範囲に収まっていることを前提に関数が記述されている場合などです。このような関数に範囲外の値が渡されると、プログラムの動作全体がおかしくなる可能性があります。

このように、システムの内部ロジックの確認が必要になる場合のために、X/Openでは**assert**マクロが用意されています。このマクロを使うと、仮定が正しいかどうかをテストし、正しくなければプログラムを停止することができます。

```
#include <assert.h>

void assert(int expression)
```

assertマクロは、指定された式を評価し、その値が0でなければ標準エラー出力に診断情報を書き込み、**abort**を呼び出してプログラムを終了します。

assertマクロはヘッダーファイル**assert.h**で定義されており、**NDEBUG**の定義によって処理

が変わるようになっています。ヘッダーファイルの処理時にNDEBUGが定義されている場合、assertはなにもしません。つまり、コンパイル時に-DNDEBUGを指定することでアサーションを無効にすることができます。別の方法として、各ソースファイルでassert.hをインクルードする前に、次のような行を記述しておくこともできます。

```
#define NDEBUG
```

9.5.1 assertの問題

このような使い方をする点は、実はassertの問題でもあります。テスト中はassertを使用していても、正式なコードでassertを無効にすると、テスト中のコードに比べて正式なコードの安全性のチェックが手薄になります。assertを正式なコードで有効にしておくことは、通常は選択肢としては考えられません。ユーザーがプログラムを実行したときにassert failedという不親切なエラーメッセージが表示され、プログラムが停止してしまう可能性があるからです。むしろ、エラーをトラップする独自のルーチンを記述してアサーションをチェックできるようにしたほうがスマートです。また、適切に記述されたルーチンなら、正式なコードで無効にしなくてもすみます。

assertでは、指定する式が副作用を持たないように注意する必要があります。たとえば、なんらかの処理を行う関数呼び出しを使うと、assertを無効にした正式なコードでは期待した処理が行われなくなってしまいます。

次に示すサンプルプログラムは、正の値を必要とする関数を定義しているassert.cです。assertマクロを使って引数が不正でないかどうかチェックしています。

例題 assert

ヘッダーファイルassert.hをインクルードし、引数が正の値かどうかをチェックして平方根を返す関数を記述します。

```
#include <stdio.h>
#include <math.h>
#include <assert.h>

double my_sqrt(double x)
{
    assert(x >= 0.0);
    return sqrt(x);
}

int main()
{
    printf("sqrt +2 = %g\n", my_sqrt(2.0));
```

```
    printf("sqrt -2 = %g\n", my_sqrt(-2.0));
    exit(0);
}
```

プログラムを実行すると、不正な値が渡されたときにアサーションが失敗します。

```
$ cc -o assert assert.c -lm
$ assert
sqrt +2 = 1.41421
assert: assert.c:7: my_sqrt: Assertion `x >= 0.0' failed.
IOT trap/Abort
$
```

解説

負の数を指定して`my_sqrt`関数を呼び出すと、アサーションは失敗します。`assert`マクロは、アサーションが失敗したファイルとその行番号、および失敗の原因を表示します。

プログラムはアボートで終了します。これは、`assert`が`abort`を呼び出した結果です。

`-DNDEBUG`を指定してプログラムを再コンパイルすると、アサーションは取り除かれ、`my_sqrt`から`sqrt`を呼び出したときに返される値は不正になります（`NaN`はnot-a-numberの意）。

```
$ cc -o assert -DNDEBUG assert.c -lm
$ assert
sqrt +2 = 1.41421
sqrt -2 = NaN
$
```


9.6 メモリデバッグ

特定困難なバグが混入しやすい領域として、動的なメモリの割り当てがあります。`malloc`と`free`を使ってメモリを割り当てるプログラムを記述する場合には、割り当てられたブロックをきちんと管理し、解放したブロックを使わないようにすることが大切です。

一般に、メモリブロックは`malloc`によって割り当て、これをポインタ変数に格納します。しかし、ポインタ変数が変更され、該当するメモリブロックを指すポインタがほかに存在しなければ、メモリブロックにアクセスすることはできなくなります。この状態がいわゆる**メモリリーク**であり、プログラムのサイズが大きくなる原因になります。大量のメモリリークが発生すると、システムの処理能力は低下し、メモリ不足におちいります。

また、割り当てられたブロックを超えて（またはブロックの先頭よりも前の領域に）書き込み

を行うと、mallocライブラリがメモリの割り当てを管理するために使っているデータ構造が破壊される可能性があります。もしデータ構造が破壊されれば、以後mallocやfreeを呼び出したときにセグメンテーション違反が発生し、プログラムはクラッシュします。この場合、問題の箇所がどこかを正確に特定することはしばしば非常に困難です。クラッシュの原因は、ずっと以前に行われた操作である可能性があるからです。

さいわい、この2つの問題を解決するのに役立つツールが、商用のものやフリーのものを含めていくつか存在しています。たとえば、mallocとfreeは、独自バージョンのものが多数提供されています。これらの中には、追加のコードによってメモリの割り当てや解放をチェックし、ブロックが2度解放されるなどの不正な使用を監視するものがあります。

9.6.1 Electric Fence

Bruce Perensが作成したElectric Fenceライブラリは、UNIXの仮想メモリ機能を使って、mallocとfreeが使うメモリを保護します。Electric Fenceは、プログラムがメモリを破壊する時点でプログラムを停止することを目的としています。

例題 Electric Fence

次のefence.cは、mallocを使ってメモリブロックを割り当て、割り当てたブロックを超えて書き込みを行います。

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    char *ptr = (char *) malloc(1024);
    ptr[0] = 0;

    /* Now write beyond the block */
    ptr[1024] = 0;
    exit(0);
}
```

プログラムをコンパイルして実行すると、プログラムはなんの問題もなく終了します。しかし、メモリ領域になんらかのごみが残り、あとでトラブルに見舞われる可能性があります。

```
$ cc -o efence efence.c
$ efence
$
```

今度は、同じプログラムをElectric Fenceのライブラリlibefence.aとリンクしてみましょ

う。プログラムを実行すると、問題があることがわかります。

```
$ cc -o efence efence.c libefence.a
$ efence

  Electric Fence 2.0.5 Copyright (C) 1987-1995 Bruce Perens.
Segmentation fault
$
```

デバッガからプログラムを実行すると、問題の箇所を特定することができます。

```
$ cc -g -o efence efence.c libefence.a
$ gdb efence
(gdb) run
Starting program: /usr/neil/efence

  Electric Fence 2.0.5 Copyright (C) 1987-1995 Bruce Perens.

Program received signal SIGSEGV, Segmentation fault.
0x80488d4 in main () at efence.c:11
11        ptr[1024] = 0;
(gdb)
```

解説

Electric Fenceは、**malloc**とその関連関数を、プロセッサの仮想メモリ機能を使うバージョンに置き換え、不正なメモリアクセスを検出します。不正なメモリアクセスが行われると、セグメンテーション違反を示すシグナルが発生し、プログラムは停止させられます。

9.6.2 Purify

PurifyはPure Software（現在はRational Software）の商用ベースの製品で、同社が特許を持つObject Code Insertion（オブジェクトコード挿入）テクノロジーを利用し、プログラムを修正してメモリアクセスに関する問題を発見します。

Purifyでは、プログラムに埋め込まれたポインタアクセスチェックコードによって、初期化されていないメモリからの読み取りや割り当てられていないメモリブロックへの書き込み、メモリリークなどを検出することができます。**Purify**はHPシステム、SparcベースのSunワークステーションなどで利用することができます。残念ながらLinux用のバージョンはまだありませんが、今後状況が変わる可能性もあります。

商用ソフトウェア開発メーカーは、**Purify**を使うことでデバッグ時間を大幅に短縮することができます。**Purify**の非常に優れている点は、チェック用コードを直接オブジェクトコードに挿入でき、システムライブラリを再コンパイルする必要がないことです。

9.6.3 Checker

Tristan Gingoldが開発したCheckerは、コンパイラバックエンドとCライブラリに修正を加えたツールで、Purifyが対象としている問題の多くを検出することができます。Checkerが特に便利なのは、メモリリークを検出できる点です。

Checkerを利用するにはプログラムの再コンパイルが必要になります。ただし、Linuxではほとんどのソフトウェアのソースコードを入手できるので、この点は大きな制約にはならないでしょう。

例題 Checker

次のchecker.cは、メモリを割り当て、初期化されていない部分からの読み取りと割り当てられた部分を超えた書き込みを行い、最後にメモリにアクセスできなくするサンプルプログラムです。

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    char *ptr = (char *) malloc(1024);
    char ch;

    /* Uninitialized read */
    ch = ptr[0];

    /* Write beyond the block */
    ptr[1024] = 0;

    /* Orphan the block */
    ptr = 0;
    exit(0);
}
```

Checkerは、コンパイラコマンドをcheckergccに置き換えるだけで利用できます。checkergccは、適切なコンパイラバージョンを呼び出して特別なChecker用ライブラリとリンクするためのドライバプログラムです。

checkergccでプログラムをコンパイルして実行すると、たくさんの問題が報告されます。

```
$ checkergcc -o checker checker.c
$ checker
Checker 0.8 (i386-linux) Copyright (C) 1996 Tristan Gingold.
This program has been compiled with `checkergcc' or `checkerg++'.
Checker is a memory access detector.
Checker is distributed in the hope that it will be useful,
```

```
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.
For more information, set CHECKEROPTS to `--help'
From Checker (pid:20795): `checker' is running (Wed Jan  7 07:23:25 1998)

From Checker (pid:20795): (ruh) read uninitialized byte(s) in a block.
When Reading 1 byte(s) at address 0x0806335c, inside the heap (sbrk).
0 bytes into a block (start: 0x806335c, length: 1024, mdesc: 0x0).
The block was allocated from:
        pc=0x08051940 in malloc() at ./l-malloc/malloc.c:251
        pc=0x080481df in main() at checker.c:6
        pc=0x0804810c in _start() at :0
Stack frames are:
        pc=0x08048200 in main() at checker.c:10
        pc=0x0804810c in _start() at :0
From Checker (pid:20795): (bvh) block bounds violation in the heap.
When Writing 1 byte(s) at address 0x0806375c, inside the heap (sbrk).
0 bytes after a block (start: 0x806335c, length: 1024, mdesc: 0x0).
The block was allocated from:
        pc=0x08051940 in malloc() at ./l-malloc/malloc.c:251
        pc=0x080481df in main() at checker.c:6
        pc=0x0804810c in _start() at :0
Stack frames are:
        pc=0x08048225 in main() at checker.c:13
        pc=0x0804810c in _start() at :0
```

不正な読み取りと書き込みがチェックされ、該当するメモリブロックとその場所が表示されます。デバッガを使えば、問題の箇所でプログラムを停止することができます。

Checkerには、特定の種類のエラーやメモリリークの検出を除外するなど、数多くのオプションがあります。サンプルプログラムでメモリリークを検出する場合には、環境変数**CHECKEROPTS**を通じてこうしたオプションのうちの1つを指定します。プログラムの終了時のメモリリークをチェックするには、次のように**-D=end**を指定します。

```
$ CHECKEROPTS=-D=end checker

...

From Checker (pid:21062): (gar) garbage detector results.
There is 1 leak and 0 potentiel leak(s).
Leaks consume 1024 bytes (1 KB) / 131471 KB.
( 0.00% of memory is leaked.)
Found 1 block(s) of size 1024.
Block at ptr=0x806363c
        pc=0x08051900 in malloc_1() at ./l-malloc/malloc.c:233
        pc=0x080481df in main() at checker.c:6
        pc=0x0804810c in _start() at :0
```

解説

`checkergcc`コンパイラを使ってプログラムをコンパイルすると、ポインタ参照をチェックするコードが追加されます。割り当てられたメモリブロックへのアクセスが不正な場合には、`Checker`からメッセージが表示されます。プログラムの終了時にはガベージコレクションルーチンが実行され、割り当てられたメモリで解放されていないものがあるかどうかが調べられます。該当するブロックがある場合には、そのブロックに関する情報が表示されます。

9.7 この章のまとめ

この章では、デバッグのためのツールとテクニックについて説明しました。UNIX、特にLinuxでは、プログラムの欠陥の除去に役立つ強力なツールを利用することができます。

具体的なデバッグの手順として、`gdb`を使ってプログラムからバグを取り除く方法を示し、`cflow`、`lclint`などの静的な分析ツールを紹介しました。

最後に、動的なメモリ割り当てに関連する問題を取り上げ、Electric Fenceや`Checker`といったユーティリティを使って問題を特定する方法を示しました。

メモ

この章で取り上げたユーティリティプログラムのほとんどは、インターネット上のFTPサーバーなどから入手することができます。筆者は主に`ftp://sunsite.unc.edu/`のLinuxアーカイブを利用しました。なお、プログラムによっては著作権の扱いに注意が必要なものもありますから、付属のドキュメントをよく読んでください。

第10章

プロセスとシグナル

UNIXコンピュータシステムのほとんどすべての動作は、UNIXオペレーティング環境の基本部分を形作るプロセスとシグナルによって制御されています。UNIXがプロセスをどのように管理しているかを理解することは、システムプログラマにとっても非常に大切です。

この章では、Linux環境でのプロセスの扱いについて取り上げ、ある与えられた時点でコンピュータが実行していることを正確に把握する方法を示します。また、プログラムの中からほかのプロセスの起動や停止を行う方法、プロセスでメッセージをやりとりする方法、ゾンビプロセスの発生を回避する方法も示します。取り上げる項目は次のとおりです。

- プロセスの構造、種類、スケジューリング
- 新しいプロセスの起動方法
- 親プロセス、子プロセス、ゾンビプロセス
- シグナルとその使い方

10.1 プロセスとは

X/Open仕様では、1つの制御スレッドが実行されるアドレス空間と、そのために必要なシステムリソースのことを**プロセス**と定義しています。

プロセスは、基本的には実行中の1つのプログラムです。UNIXなどのマルチタスクオペレーティングシステムでは、多くのプログラムを同時に実行でき、実行中のプログラムの各インスタンスが1つのプロセスを構成します。X Window Systemなどのウィンドウシステムを使うと、複数のプログラムが同時に実行されるようすを実際に目で確かめることができます。グラフィカルユーザーインタフェースを提供するX Window Systemでは、Microsoft Windows同様、それぞれ独自のウィンドウを持つ複数のプログラムを同時に実行できます。

UNIXはマルチユーザーシステムであり、複数のユーザーが同時にシステムにアクセスできます。各ユーザーは、それぞれ複数のプログラムを実行することができます。システム自体も、ほかのプログラムを実行してシステムリソースを管理し、ユーザーのアクセスを制御します。

プロセスを構成するのは、本質的にはなんらかのプログラムコード、（システムメモリの一部を占める）なんらかのデータと変数、オープンされたファイル（ファイルデスクリプタ）、そして（第4章で見たような）環境です。

一般に、UNIXシステムでは、ある時点でメモリ内に存在するプログラムコードのコピーがただ1つになるように、プロセス間でコードを共有します。このことは、共有システムライブラリにもあてはまります。

10.2 プロセスの構造

実際に、2つのプロセスがオペレーティングシステム内部でどのように配置されているかを見てみましょう。例として、**neil**と**rick**の2人のユーザーが、どちらも同時に**grep**プログラムを実行し、それぞれ別のファイルで異なる文字列を検索するとします。プログラムの実行中、使われているプロセスは次の図のようになります。

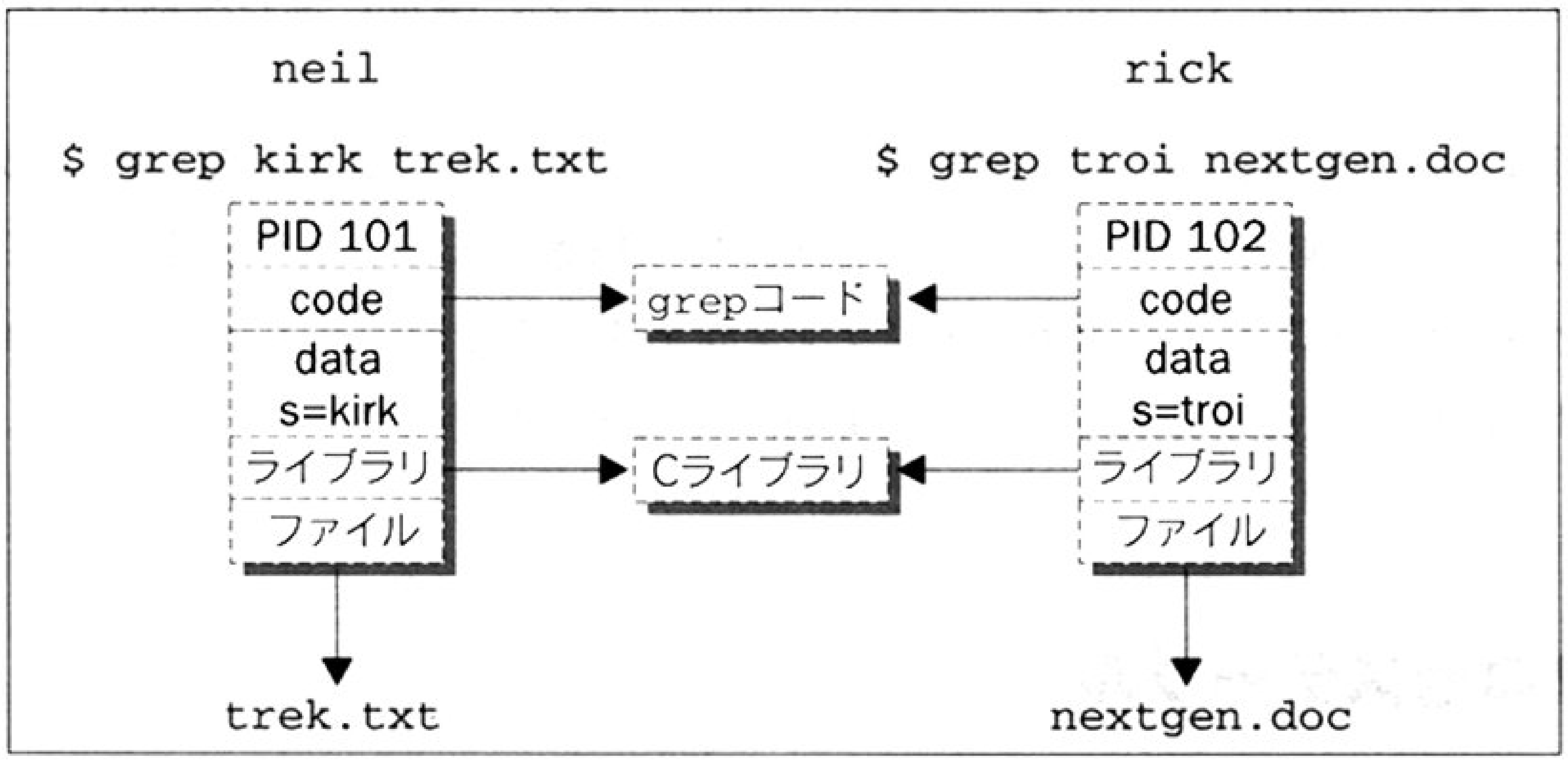


図10.1 プロセスの構造

各プロセスには、一意の番号である**プロセスID**（PID）が割り当てられます。通常、プロセスIDは2～32000の範囲の整数です。プロセスが起動されると、使われていない次の番号がプロセスIDとして選択され、範囲内の最後の整数に達した場合には、再び2から順に割り当てられていきます。上の図で、**neil**と**rick**が起動した2つのプロセスには、プロセスIDとしてそれぞれ101と102が割り当てられています。

grepコマンドによって実行されるプログラムコードは、ディスクファイルに格納されています。通常、UNIXのプロセスは、プログラムコードを保持するのに使われているメモリ領域に対しては書き込みを行いません。このため、プログラムコードは読み取り専用としてメモリにロードされます。図からわかるように、書き込みができないのでプログラムコードを安全に共有でき、物理メモリを効率的に使用できます。

システムライブラリも共有可能です。これは、たとえば**printf**を呼び出すプログラムがいくつ実行されていても、メモリ内には**printf**のコピーが1つだけ存在すればよいことを意味します。このしくみは、Microsoft Windowsのダイナミックリンクライブラリといくつかの点で共通するものがあります。

共有ライブラリを使った場合には、上の図でも明らかなように、実行プログラム**grep**を格納しているディスクファイルのサイズを小さくできるという利点もあります。

もちろん、プログラムのすべての部分を共有できるわけではありません。たとえば、プログラムで使われる変数は、プロセスごとに区別されます。上の図の例では、**grep**コマンドに渡され

た検索文字列が、プロセスごとのデータ空間に変数**s**として存在しています。これらの変数は互いに独立しており、通常はほかのプロセスから読み取ることはできません。また、2つの**grep**コマンドで使われているファイルも異なります。各プロセスでは、ファイルアクセス用の一連のファイルデスクリプタをそれぞれ独自に管理しています。

各プロセスは、関数内のローカル変数や、関数の呼び出しと関数からの復帰のために使う独自のスタック空間を持っています。また、環境空間も独自のものを持っており、各プロセスだけで使う環境変数を用意することができます。各プロセスには、このほかに、プログラムがどこまで実行されたかを示す独自のプログラムカウンタがあります。通常、これは**実行スレッド**と呼ばれます。

上の図では、プロセス、コード、データ、ファイルデスクリプタがメモリの一部を占めていますが、すでに以前の章で説明したように、UNIXでは仮想メモリシステムを使っており、コードやデータをハードディスク上の領域にスワップアウトすることができます。このため、実際には、物理メモリが許す以上に多くのプロセスを管理することができます。

10.2.1 プロセステーブル

UNIXの**プロセステーブル**は、現在ロードされているすべてのプロセスについての情報を収めたデータ構造と考えることができます。各プロセスに関する情報としては、**ps**コマンドによって表示される情報、つまりプロセスID、端末の種類、ステータス、コマンドなどがあります。オペレーティングシステムはプロセスIDによってプロセスを管理しており、プロセスIDはプロセステーブルへのインデックスとして使われます。プロセステーブルのサイズには制限があるため、システムがサポートしているプロセスの数もその制限を受けます。かつてのUNIXシステムでは、プロセス数は256に制限されていました。この制限は新しいUNIXの実装では大幅に緩和されており、現在では、一般にプロセステーブルエントリの作成に利用可能なメモリによってのみ制限を受けます。

10.2.2 プロセスの表示

psコマンドを使うと、現在どのようなプロセスが実行されているかを表示することができます。**ps**コマンドでは、自分が実行しているプロセスだけでなく、システムで実行されているすべてのプロセスについての情報を表示することができます。

次に示すのは、X Window Systemを実行しているLinuxシステム上での**ps**コマンドの出力例です。

```
$ ps
  PID TTY STAT  TIME COMMAND
```

```
 95   1 S    0:00 -bash
104   1 S    0:00 sh /usr/X11/bin/startx
108   1 S    1:42 fvwm
146   1 S    0:00 oclock
399  p0 S    0:00 -bash
587  p0 S    0:00 emacs process.txt
```

この出力例では、**emacs**エディタによるファイルの編集に関係したプロセスが表示されています。**PID**列にはプロセスIDが、**TTY**列には該当するプロセスを起動した端末が、**STAT**列には現在のステータスが、**TIME**列にはこれまでに使ったCPU時間が、また**COMMAND**列にはプロセスを起動するのに使われたコマンドが表示されています。

出力の内容を少し詳しく見ていきましょう。まず最初の行です。

```
 95   1 S    0:00 -bash
```

ログインは、仮想コンソール番号1(**TTY**列の**1**)で実行されました。これは、このマシン(PC)のコンソールです。実行中のシェルプログラムは、Linuxのデフォルトシェル**bash**です。ステータスは、休眠中(sleeping)を意味する**S**になっています。これは、次に示す**startx**の終了を待っているためです。

```
104   1 S    0:00 sh /usr/X11/bin/startx
```

X Window Systemは、**startx**コマンドによって起動されました。**startx**はXサーバーを起動するシェルスクリプトで、ほかにもいくつかのX Windowプログラムを実行します。このシェルスクリプトは、Xを終了するまで終わりません。このプロセスも休眠中になっています。

```
108   1 S    1:42 fvwm
```

fvwmはウィンドウマネージャです。ウィンドウマネージャを使うと、ほかのプログラムを起動したり、ウィンドウの配置を変えたりすることができます。

```
399  p0 S    0:00 -bash
```

このプロセスは、X Window Systemの1つのウィンドウを表しており、新しいウィンドウの作成要求に応答してウィンドウマネージャが起動したものです。シェルの**bash**は、作成された新しいウィンドウで実行されています。このシェルには、読み書き用に新しい仮想端末**/dev/ttyp0**が割り当てられており、**ps**からの出力では省略形の**p0**が使われています。シェルはこの仮想端末を使って、新しいウィンドウに出力を書き込みます。ウィンドウで文字を入力すると、入力内容はこの仮想端末を介してシェルに渡されます。

```
587  p0 S    0:00 emacs process.txt
```

これは、上で説明したシェルから起動された**emacs**エディタのセッションです。このプロセスも、入出力に仮想端末を使っています。

```
  146    1 S     0:00 oclock
```

これは、ウィンドウマネージャから起動された時計プログラムです。時計の針は1分間隔で更新されますが、現在は前回の更新と次回の更新のちょうど間で休眠中になっています。

psコマンドがデフォルトで表示するプロセスは、コンソール、シリアル回線、仮想端末など、端末を持つプロセスだけです。その他のプロセスは、端末上のユーザーと情報をやりとりすることなく実行されています。この種のプロセスとしては、共有リソースを管理するためのシステムプロセスなどがあります。**-x**オプションを指定して**ps**コマンドを実行すると、端末を持たないプロセスも表示することができます。ほかのユーザーのプロセスに関する情報を表示するには、**-a**オプションを使います。**ps**コマンドの構文や出力形式はシステムによって異なることがあるので、オプションと出力形式の詳細についてはマニュアルページを参照してください。

10.2.3 システムプロセス

Linuxシステム上で実行中のほかのプロセスについても見ておきましょう。次の出力例は、一部を省略したものです。

```
  PID TTY STAT  TIME COMMAND
    1  ?  S     0:03 init [3]
    2  ?  SW    0:00 (kflushd)
    3  ?  SW<   0:00 (kswapd)
   12  ?  S     0:00 /sbin/update
   58  ?  S     0:00 /usr/sbin/klogd
   59  ?  S     0:00 /usr/sbin/syslogd
   75  ?  S     0:00 sendmail: accepting connections on port 25
   84   3 S     0:00 /sbin/agetty 38400 tty3 linux
  106  ?  R    99:58 X :0
 5759 p2  R     0:00 ps -ax
```

この出力例には、非常に重要なプロセスが含まれています。それは次のプロセスです。

```
    1  ?  S     0:03 init [3]
```

一般に、どのプロセスもほかのプロセスから起動されます。このとき、起動したほうのプロセスを**親プロセス**と呼び、起動されたほうのプロセスを**子プロセス**と呼びます。UNIXは、起動されると1つのプログラムを実行します。これが、すべてのプロセスの祖先となる**init**です。**init**は、オペレーティングシステムのプロセスマネージャともいうべき存在です。ほかのシステムプロセスは、**init**から起動されるか、または**init**から起動されたほかのプロセスから起動されます。

ログイン処理を例にとってみましょう。**init**は、ログインに利用できる各端末に対して**getty**プログラムを起動します。**ps**コマンドの出力では、起動された**getty**プロセスは次のように表示されます。

```
   84   3 S    0:00 /sbin/agetty 38400 tty3 linux
```

gettyプロセスは、端末に**login:**プロンプトを表示し、ユーザーからのログインを促します。ユーザーがユーザー名を入力すると、**getty**は制御を**login**プログラムに引き渡します。**login**プログラムは、ユーザーの環境をセットアップし、最後にシェルを起動します。ユーザーがこのシェルを終了すると、**init**は再び**getty**プロセスを起動します。

このことから、新しいプロセスを開始し、開始したプロセスの終了を待つことがシステムにとって重要な意味を持っていることがわかります。この章の後のほうでは、プログラムの中からシステムコールの**fork**、**exec**、**wait**を使って同じ処理を行う方法を示します。

10.2.4 プロセススケジューリング

psコマンドからの出力には、**ps**コマンド自体のエントリも含まれています。

```
 5759  p2 R    0:00 ps -ax
```

これは、プロセス5759が実行状態（**R**）にあり、コマンド**ps -ax**を実行中であることを示しています。もちろん、これは上の出力を生成したプログラムです。ステータスの**R**は、実行可能であることを示しているだけで、必ずしも実際に実行されていることを意味するわけではありません。シングルプロセッサコンピュータ上では、一度に実行できるプロセスは1つだけで、ほかのプロセスは自分の順番が来るのを待っています。各プロセスが実行される時間は**タイムスライス**と呼ばれ、非常に短い時間なので、複数のプログラムが同時に実行されているかのように見えます。ステータスの**R**は、そのプログラムがほかのプロセスの終了を待機したり、入出力の完了を待機したりしているのではないことを示しています。**ps**コマンドの出力例で、ステータスの列が**R**になっているプロセスが2つ存在するのもそのためです（もうひとつのプロセスは、X Window Systemのディスプレイサーバーです）。

UNIXシステムは、プロセススケジューラを使って、次のタイムスライスをどのプロセスに割り当てるかを決定します。この決定は、プロセスの優先度に基づいて行われます。優先度についてはすでに第4章で少し触れました。基本的には、優先度の高いプロセスほどより多く実行され、ほかのプロセス（優先度の低いバックグラウンドタスクなど）はそれより少なく実行されます。UNIXでは、割り当てられたタイムスライスを超えてプロセスが実行を続けることはできません。プロセスは互いに協調することなく停止され、再開されます。このような方式のマルチタスクのことを、**プリエンプティブなマルチタスク**といいます。この点は、プログラムの側で譲歩してほかのプログ

ラムが実行できるようにする必要のあるMicrosoft Windows 3.xとは異なります。

プロセスの優先度は、**nice**コマンドで設定でき、**renice**コマンドで調整できます。優先度のデフォルトは0で、優先度の高いプロセスは負の値を持ち、優先度の低いプロセスは正の値を持ちます。**nice**コマンドは、デフォルトでは優先度の値を10だけ増やして（つまり優先度を引き下げて）プログラムを実行します。アクティブなプロセスの優先度の値は、**ps**コマンドに**-l**オプションを指定することで表示できます。その値は**NI**（nice）列に表示されます。

```
$ ps -l
 FLAGS   UID   PID  PPID PRI  NI   SIZE   RSS WCHAN       STA TTY TIME COMMAND
100000 10001  6424  5765   1   0   1952  1044  12da71      S   p2  0:00 oclock
```

oclockプログラムはデフォルトのnice値で実行されています。今度は、次のコマンドでプログラムを実行してみます。

```
$ nice oclock &
```

これで、nice値には+10が割り当てられます。実行中のプロセスの優先度を変更するには、次のように**renice**コマンドを使います。

```
$ renice 10 6462
6462: old priority 0, new priority 10
```

これで、**oclock**プログラムの実行頻度は低くなります。変更後のnice値は、**ps**コマンドで表示することができます。

```
$ ps -l
 FLAGS   UID   PID  PPID PRI  NI   SIZE   RSS WCHAN       STA TTY TIME COMMAND
100000 10001  6462  6457   0  10   1952  1044  12da71      S N p2  0:00 oclock
```

ステータスに**N**が含まれていることに注目してください。この**N**は、nice値がデフォルトではなく変更されていることを示しています。**PPID**フィールドには、親プロセスのプロセスIDが表示されます。これは、**oclock**プログラムを起動したプロセスのプロセスIDか、あるいは**oclock**プログラムを起動したプロセスがもう実行されていない場合には、**init**のプロセスID（つまり1）です。

プロセスが多く実行されれば、それだけそのプロセスが得る優先度も低くなります。UNIXのスケジューラは、優先度に基づいてどのプロセスを実行するかを決定します。もちろん、具体的な実装がどのようになっているかは細かな部分で異なる可能性がありますが、一般には優先度の高いプロセスほど多く実行されます。より優先度の高い実行可能プロセスがほかにある場合、優先度の低いプロセスがまったく実行されないといったこともあります。

10.3 新しいプロセスの起動

ライブラリ関数の**system**を使うと、プログラムの中から別のプログラムを呼び出して新しいプロセスを作成することができます。

```
#include <stdlib.h>

int system (const char *string);
```

system関数は、**string**に指定されたコマンドを実行し、このコマンドの実行が終了するのを待ちます。コマンドは、シェルで次のコマンドを入力したときと同様に実行されます。

```
$ sh -c string
```

system関数は、コマンドを実行するシェルを起動できなかった場合には127を返し、その他のエラーが発生した場合には-1を返します。それ以外の場合には、コマンドの終了コードを返します。

例題 system

1 **system**関数を使って**ps**を実行するプログラム**system.c**を作成します。プログラムとしての利用価値はあまりありませんが、後のサンプルプログラムではこのテクニックの応用例を示します。

```
#include <stdlib.h>
#include <stdio.h>

int main()
{
    printf("Running ps with system\n");
    system("ps -ax");
    printf("Done.\n");
    exit(0);
}
```

2 プログラムをコンパイルして実行すると、次のように出力されます。

```
$ system
Running ps with system
  PID TTY STAT  TIME COMMAND
    1  ?  S    0:03 init [3]
    2  ?  SW   0:00 (kflushd)
    3  ?  SW<  0:00 (kswapd)
   12  ?  S    0:00 /sbin/update
```

10

```
...
 6462  p2 S N  0:00 oclock
 6574  p3 S    0:00 system
 6575  p3 R    0:00 ps -ax
Done.
```

3 **system**関数は、指定されたプログラムをシェルを使って起動するので、プログラムをバックグラウンドで実行することもできます。**system.c**を次のように変更します。名前は**system2.c**とします。

```
system("ps -ax &");
```

プログラムをコンパイルして実行すると、今度は次のように出力されます。

```
$ system2
Running ps with system
Done.
$   PID TTY STAT  TIME COMMAND
    1  ?  S     0:03 init [3]
    2  ?  SW    0:00 (kflushd)
    3  ?  SW<   0:00 (kswapd)
   12  ?  S     0:00 /sbin/update
...
 6462  p2 S N  0:00 oclock
 6593  p3 R    0:00 ps -ax
```

解説

最初のサンプルプログラムでは、**"ps -ax"**を指定して**system**を呼び出し、**ps**コマンドを実行しています。このプログラムでは、**ps**コマンドが終了したときに**system**の呼び出しから戻ります。**system**は便利な関数ですが、制約もあります。具体的には、**system**を呼び出して起動したプロセスが終了するまで待機しなければならないので、ほかのタスクを実行することができません。

2番目のサンプルプログラムでは、**system**の呼び出しは、シェルコマンドが終了すると同時に戻ります。今度はバックグラウンドでプログラムを実行するように要求しているので、シェルは**ps**プログラムを起動すると同時に戻ります。これは、ちょうど次のようにしてシェルプロンプトでコマンドを実行した場合と同じです。

```
$ ps -ax &
```

systemの呼び出しから戻ると、**system2**プログラムは**Done.**というメッセージを表示し、**ps**コマンドが画面に出力を送る前に終了します。**ps**コマンドからの出力は、**system2**の終了後に表示されたシェルプロンプトの後に表示されています。

このようなプログラムの動作は混乱を招きやすく、ユーザーにとっては不親切です。プロセスを上手に扱うには、その動作をもっと細かく指定する必要があります。以下では、プロセスを作成するための低水準インタフェース**exec**について見ていきます。

◆ プロセスイメージの置き換え

execで始まる名前を持つ関数はたくさんあります。これらの関数は、プロセスの起動方法や引数の指定方法が互いに異なります。

```
#include <unistd.h>

char **environ;

int execl(const char *path, const char *arg0, ...,  (char *)0);
int execlp(const char *path, const char *arg0, ...,  (char *)0);
int execle(const char *path, const char *arg0, ...,  (char *)0, const
char *envp[]);
int execv(const char *path, const char *argv[]);
int execvp(const char *path, const char *argv[]);
int execve(const char *path, const char *argv[], const char *envp[]);
```

exec族の関数は、指定された引数に基づいて作成される別のプロセスで現在のプロセスを置き換えます。引数**path**に指定されたプログラムは、現在実行されているプログラムの代わりに実行するプログラムコードとして使われます。**execl**関数群の場合、**arg0**、**arg1**など、ヌルポインタまでの引数が新しいプログラムに引数として渡されます。一方、**execv**関数群の場合には、文字列配列へのポインタ**argv**によって新しいプログラムの引数を指定します。どちらの場合にも、指定された引数は**main**関数に渡される配列**argv**になり、新しいプログラムが起動されます。

末尾に**p**が付く関数は、環境変数**PATH**に基づいて新しいプログラムの実行可能ファイルを探します。目的の実行可能ファイルがパス上にない場合には、ディレクトリを含む絶対ファイル名を関数のパラメータとして指定する必要があります。

グローバル変数**environ**は、新しいプログラムの環境に値を渡すために使うことができます。**execle**と**execve**には、新しいプログラムの環境として使う文字列配列を渡すための追加の引数があります。

exec族の関数を使って前のサンプルプログラムのように**ps**コマンドを実行する場合、次のようなさまざまな方法が可能です。

```
#include <unistd.h>

/* Example of an argument list */
/* Note that we need a program name for argv[0] */
const char *ps_argv[] =
    {"ps", "-ax", 0};
```

```
/* Example environment, not terribly useful */
const char *ps_envp[] =
    {"PATH=/bin:/usr/bin", "TERM=console", 0};

/* Possible calls to exec functions */
execl("/bin/ps", "ps", "-ax", 0);           /* assumes ps is in /bin */
execlp("ps", "ps", "-ax", 0);               /* assumes /bin is in PATH */
execle("/bin/ps", "ps", "-ax", 0, ps_envp); /* passes own environment */

execv("/bin/ps", ps_argv);
execvp("ps", ps_argv);
execve("/bin/ps", ps_argv, ps_envp);
```

例題 execlp

execlp関数を使ってサンプルプログラムを修正してみましょう。新しいプログラムの名前は**pexec.c**とします。

```
#include <unistd.h>
#include <stdio.h>

int main()
{
    printf("Running ps with execlp\n");
    execlp("ps", "ps", "-ax", 0);
    printf("Done.\n");
    exit(0);
}
```

プログラムを実行すると、**ps**コマンドからの出力は表示されますが、**Done.**というメッセージは表示されません。また、**ps**からの出力には**pexec**のプロセスは見あたりません。

```
$ pexec
Running ps with execlp
  PID TTY STAT  TIME COMMAND
    1  ?  S    0:03 init [3]
    2  ?  SW   0:00 (kflushd)
    3  ?  SW<  0:00 (kswapd)
   12  ?  S    0:00 /sbin/update
...
 6462  ?  S N  0:00 oclock
 6636 p3 R     0:00 ps -ax
```

解説

このプログラムは、最初のメッセージを表示した後、`execlp`を呼び出します。`execlp`は、環境変数`PATH`で指定されたディレクトリから`ps`という名前のプログラムを探します。次に、`pexec`プログラムの代わりに`ps`プログラムを実行します。`ps`プログラムは、次のシェルコマンドを入力したときと同様に実行されます。

```
$ ps -ax
```

`ps`プログラムが終了すると、新しいシェルプロンプトが表示されます。`pexec`プログラムには戻らないので、2番目のメッセージは表示されません。新しいプロセスのプロセスIDはもとのプロセスのプロセスIDと同じです。また、親プロセスのPIDとnice値ももとのプロセスのものと同じです。つまり、実行中のプログラム（`pexec`）は、`exec`の呼び出しで指定された新しい実行可能ファイルから新しいコードの実行を開始しています。

`exec`関数群から起動されるプロセスの引数リストと環境との合計サイズには上限があることに注意する必要があります。この上限は`ARG_MAX`で与えられ、Linuxシステムの場合には128KBです。ほかのシステムでは上限がこれより大幅に少ないことがあり、このことが問題になる場合もあります。POSIX仕様では、`ARG_MAX`の最小値を4096バイトと規定しています。

`exec`族の関数は、エラーが発生した場合を除いて戻ることはありません。エラーが発生した場合には、`errno`にエラーの内容を示す値を設定し、-1を返します。

`exec`によって起動された新しいプロセスは、もとのプロセスから多くのものを引き継ぎます。特に、オープンされたファイルデスクリプタは、`FD_CLOEXEC`フラグが設定されている場合を除いて、新しいプロセスでもオープンされたままになります（`FD_CLOEXEC`フラグについては、第3章の`fcntl`システムコールの説明を参照してください）。もとのプロセスでオープンされていたディレクトリストリームがある場合、これらのディレクトリストリームはクローズされます。

◆ プロセスイメージの複製

複数のプロセスを使って一度にたくさんの作業を実行するには、`exec`のように現在の実行スレッドを置き換えるのではなく、ちょうど`init`が行うように、完全に独立したプロセスをプログラムの中から作成する必要があります。

このような場合、新しいプロセスを作成するには`fork`を呼び出します。このシステムコールは、現在のプロセスを複製し、現在のプロセスとほとんど同じ属性を持つ新しいエントリをプロセステーブルに作成します。新しいプロセスはもとのプロセスとほぼ同一であり、同じコードを実行しますが、独自のデータ空間、環境、ファイルデスクリプタを持ちます。目的に応じた新しいプロセスを作成するには、`exec`のほかに、この`fork`があれば十分です。

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

forkシステムコールは、新しい子プロセスを作成します。作成される新しいプロセスは、固有のプロセスIDを持つ点、および親プロセスのID（PPID）として呼び出し側のプロセスのPIDを持つ点を除けば、呼び出し側のプロセスと同じです。新しい子プロセスは、親のデータ空間（変数）、およびオープンされたファイルデスクリプタとディレクトリストリームのコピーを継承します。そのほかの細かな違いについては、**fork**のマニュアルページを参照してください。

親の側での**fork**の呼び出しは、新しい子プロセスのPIDを返します。新しいプロセスの側ではもとのプロセスとまったく同様に実行が続けられますが、子プロセスでの**fork**の呼び出しは0を返す点が異なります。このことを利用すれば、どちらが親プロセスでどちらが子プロセスかを知ることができます。

forkは、失敗すると-1を返します。通常、**fork**が失敗するのは、親プロセスが持つことのできる子プロセスの数（**CHILD_MAX**）が制限されていることが原因です。この場合、**errno**には**EAGAIN**が設定されます。プロセステーブルにエントリを作成するためのスペースが不足していることが原因で**fork**が失敗した場合、**errno**には**ENOMEM**が設定されます。

一般に、**fork**は次のようにして使います。

```
pid_t new_pid;

new_pid = fork();

switch(new_pid) {
case -1 : /* Error */
    break;
case 0  : /* We are child */
    break;
default : /* We are parent */
    break;
}
```

例題 fork

簡単なサンプルプログラム**fork.c**を作成してみましょう。

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main()
{
    pid_t pid;
```

```
    char *message;
    int n;

    printf("fork program starting\n");
    pid = fork();
    switch(pid)
    {
    case -1:
        exit(1);
    case 0:
        message = "This is the child";
        n = 5;
        break;
    default:
        message = "This is the parent";
        n = 3;
        break;
    }

    for(; n > 0; n--) {
        puts(message);
        sleep(1);
    }
    exit(0);
}
```

このプログラムは2つのプロセスとして実行されます。子プロセスはメッセージを5回表示します。もとのプロセス、つまり親プロセスはメッセージを3回だけ表示します。次に、プログラムの実行例を示します。


```
$ fork
fork program starting
This is the parent
This is the child
This is the parent
This is the child
This is the parent
This is the child
$ This is the child
This is the child
```

親プロセスは、子プロセスがメッセージをすべて表示し終える前に終了しています。このため、シェルプロンプトが子プロセスからの出力に混じって表示されています。

解説

forkが呼び出されると、プログラムは2つの独立したプロセスに分かれます。forkからの戻り値が0でなければ、親プロセスです。このことを利用して、1秒おきに表示するメッセージの回数を設定しています。

10.3.1 プロセスの待機

forkで子プロセスを開始すると、開始された子プロセスは独立して実行されます。しかし、場合によっては子プロセスが終了したかどうかを知りたいことがあります。たとえば、前のサンプルプログラムでは、親プロセスが子プロセスの前に終了し、子プロセスのほうは実行を続けているために、出力が乱れてしまっています。このような場合、処理を続行する前に**wait**を呼び出すと、親プロセスは子プロセスの終了を待つことができます。

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *stat_loc);
```

waitシステムコールを呼び出すと、親プロセスは子プロセスのいずれかが死ぬか、または停止されるまで待機します。**wait**は、ステータス情報が利用可能な子プロセスのPIDを返します。通常、このPIDは、終了した子プロセスのPIDです。親プロセスの側では、ステータス情報を調べることで子プロセスの終了ステータス（**main**関数からの戻り値または**exit**に渡された値）を知ることができます。**stat_loc**がヌルポインタでなければ、**stat_loc**によって示される場所にステータス情報が書き込まれます。

ステータス情報を調べるには、**sys/wait.h**で定義されている次のマクロを使います。

表10.1 子プロセスのステータスを調べるマクロ

マクロ	定義
WIFEXITED(stat_val)	子プロセスが正常に終了した場合は0以外
WEXITSTATUS(stat_val)	**WIFEXITED**が0以外の場合は子プロセスの終了コード
WIFSIGNALED(stat_val)	捕捉されないシグナルによって子プロセスが終了した場合は0以外
WTERMSIG(stat_val)	**WIFSIGNALED**が0以外の場合はシグナル番号
WIFSTOPPED(stat_val)	子プロセスがシグナルによって停止している場合は0以外
WSTOPSIG(stat_val)	**WIFSTOPPED**が0以外の場合はシグナル番号

例題 wait

1 前のサンプルプログラムを少し修正し、子プロセスの終了を待って終了ステータスを調べることができるようにします。プログラムの名前は**wait.c**とします。

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>

int main()
```

```
{
    pid_t pid;
    char *message;
    int n;
    int exit_code;

    printf("fork program starting\n");
    pid = fork();
    switch(pid)
    {
    case -1:
        exit(1);
    case 0:
        message = "This is the child";
        n = 5;
        exit_code = 37;
        break;
    default:
        message = "This is the parent";
        n = 3;
        exit_code = 0;
        break;
    }

    for(; n > 0; n--) {
        puts(message);
        sleep(1);
    }
```

2 次のコードは、子プロセスの終了を待機する部分です。

```
    if(pid) {
        int stat_val;
        pid_t child_pid;

        child_pid = wait(&stat_val);

        printf("Child has finished: PID = %d\n", child_pid);
        if(WIFEXITED(stat_val))
            printf("Child exited with code %d\n", WEXITSTATUS(stat_val));
        else
            printf("Child terminated abnormally\n");
    }
    exit (exit_code);
}
```

プログラムを実行すると、親プロセスが子プロセスの終了を待っていることがわかります。出力も乱れず、意図したとおり終了コードを得ることができます。

```
$ wait
fork program starting
This is the parent
```

```
This is the child
This is the parent
This is the child
This is the parent
This is the child
This is the child
This is the child
Child has finished: PID = 410
Child exited with code 37
$
```

解説

親プロセス（**fork**呼び出しからの戻り値が0以外のプロセス）は、**wait**システムコールを使って、子プロセスのステータス情報が利用可能になるまで実行を一時停止します。子プロセスが終了すると、ステータス情報が利用可能になります。このプログラムでは、子プロセスの終了コードを37としています。子プロセスが終了すると、親プロセスは実行を続け、**wait**からの戻り値を使って、子プロセスが正常に終了したかどうかを調べ、ステータス情報から終了コードを取り出します。

10.3.2 ゾンビプロセス

forkを使ったプロセスの作成は非常に有益なテクニックですが、子プロセスの最後をきちんと見とらなければなりません。子プロセスが終了した場合でも、親プロセスとの関係は、親プロセスが正常に終了するか、または**wait**を呼び出すまで存続しています。つまり、子プロセスが終了しても、プロセステーブル内の子プロセスのエントリが即座に解放されるわけではなく、子プロセスは（アクティブではないものの）まだシステム内に存在しています。これは、親プロセスが**wait**を呼び出したときに備えて終了コードを保存しておく必要があるためです。このような状態の子プロセスのことを**ゾンビプロセス**と呼びます。

サンプルプログラム**fork.c**でメッセージを表示する回数を変更すると、ゾンビプロセスが作成されるのを確かめることができます。子プロセスのほうが親プロセスよりメッセージの表示回数が少なければ、子プロセスのほうが最初に死に、子プロセスは親プロセスが終了するまでゾンビプロセスとなります。

例題 ゾンビプロセス

サンプルプログラム**fork.c**で親プロセスと子プロセスのメッセージの表示回数を入れ替え、修正後の新しいプログラムを**fork2.c**とします。

```
switch(pid)
{
case -1:
    exit(1);
case 0:
    message = "This is the child";
    n = 3;
    break;
default:
    message = "This is the parent";
    n = 5;
    break;
}
```

解説

fork2 &としてプログラムを実行し、子プロセスが終了した後、親プロセスが終了する前に**ps**プログラムを実行すると、次のように表示されます。(Linuxシステム以外の場合には、一般に**<zombie>**ではなく**<defunct>**と表示されます。)

```
  PID TTY STAT  TIME COMMAND
14045  p0 Z    0:00 (fork2 <zombie>)
```

親が正常に終了しなかった場合には、PID 1のプロセス、つまり**init**が自動的に子プロセスの親になります。子プロセスはもう実行されておらず、ゾンビになりますが、親プロセスが異常終了したために**init**がその面倒をみます。ゾンビは、次にプロセステーブルが処理されるまでプロセステーブルにとどまります。プロセステーブルが大きいと、テーブルの処理に時間がかかることがあります。**init**がきちんと処理を済ませるまで、ゾンビプロセスは貴重なリソースを消費することになるので、ゾンビプロセスの発生は避ける必要があります。

また、不要なファイルがオープンされたままになることも避けなければなりません。利用可能なファイルデスクリプタの数は限られているので、プログラムの終了時にファイルをクローズしないと、リソースの無駄使いになります。

子プロセスを待機するときに使用できるシステムコールには、もうひとつ**waitpid**があります。**waitpid**を使うと、特定のプロセスの終了を待つことができます。

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

引数**pid**には、目的の子プロセスのプロセスIDを指定します。-1を指定すると、任意の子プロセスの情報が返されます。**wait**の場合と同様、**stat_loc**がヌルポインタでなければ、**stat_loc**によって示される場所にステータス情報が書き込まれます。引数**options**は、

waitpidの動作を制御するのに使います。最も便利なオプションはWNOHANGです。WNOHANGを指定すると、waitpidを呼び出しても呼び出し側の実行は停止されません。このオプションは、いずれかの子プロセスが終了したかどうかを調べ、終了していない場合には実行を続けるときに便利です。その他のオプションについては、waitのマニュアルページを参照してください。

waitpidを使って、親プロセスの側で特定の子プロセスが終了したかどうかを定期的に調べるには、次のようにします。

```
waitpid(child_pid, (int *) 0, WNOHANG);
```

子プロセスが終了または停止されていなければ戻り値は0になり、終了または停止されていれば戻り値はchild_pidになります。waitpidは、エラーが発生すると -1を返し、エラーの内容を示す値をerrnoに設定します。エラーが発生する原因には、子プロセスが存在しない（errnoはECHILD）、呼び出しがシグナルによって割り込まれた（EINTR）、オプション引数が不正である（EINVAL）などがあります。

10.3.3 入出力のリダイレクト

forkとexecの呼び出しでは、オープンされたファイルデスクリプタがそのまま引き渡されますが、このことを利用するとプログラムの動作を変更することができます。たとえば、通常はフィルタとして動作し、標準入力を読み取ってなんらかの加工をしてから標準出力に書き込むプログラムがあるとします。

例題 入出力のリダイレクト

1 まず、すべての文字を大文字に変換する簡単なフィルタプログラムupper.cを作成します。

```
#include <stdio.h>
#include <ctype.h>

int main()
{
    int ch;
    while((ch = getchar()) != EOF) {
        putchar(toupper(ch));
    }
    exit(0);
}
```

プログラムを実行すると、入力内容が読み取られて変換されます。

```
$ upper
hello THERE
HELLO THERE
^D
$
```

もちろん、シェルのリダイレクトを使ってファイルの内容を大文字に変換することもできます。

```
$ cat file.txt
this is the file, file.txt, it is all lower case.
$ upper < file.txt
THIS IS THE FILE, FILE.TXT, IT IS ALL LOWER CASE.
```

2 では、このフィルタをほかのプログラムの中で使用できないでしょうか。次の`useupper.c`は、ファイル名を引数に取り、呼び出し方が適切ではない場合にはエラーメッセージを表示します。

```
#include <unistd.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    char *filename;

    if(argc != 2) {
        fprintf(stderr, "usage: useupper file\n");
        exit(1);
    }

    filename = argv[1];
```

3 引数のチェック後、標準出力を再オープンし、エラーが発生していないことを確認してから`execl`を使って`upper`を呼び出します。

```
    if(!freopen(filename, "r", stdin)) {
        fprintf(stderr, "could not redirect stdin to file %s\n", filename);
        exit(2);
    }

    execl("./upper", "upper", 0);
```

4 `execl`は現在のプロセスを置き換えます。したがって、エラーが発生しなければ、コードの残りの部分は実行されません。

```
    fprintf(stderr, "could not exec upper!\n");
    exit(3);
}
```

解説

upper自体はファイル名を引数に取りませんが、このプログラムを使うことで大文字に変換するファイルを指定できるようになります。このプログラムの動作にはupperのソースコードは必要ありません。つまり、このプログラムのような方法を使えば、任意のプログラムを実行することができます。

```
$ useupper file.txt
THIS IS THE FILE, FILE.TXT, IT IS ALL LOWER CASE.
```

useupperプログラムは、freopenを使って標準入力をクローズし、ファイルストリームstdinをプログラムの引数で指定されたファイルに関連付けます。execlを呼び出しても、オープンされたファイルデスクリプタはそのままになるので、upperプログラムは次のシェルコマンドを実行した場合とまったく同じように動作します。

```
$ upper < file.txt
```

10.4 スレッド

UNIXのプロセスは協調することができます。つまり、互いにメッセージをやりとりしてほかのプロセスの動作を中断したりすることができます。また、プロセス間でメモリセグメントを共有することもできます。ただし、オペレーティングシステム内部では、複数のプロセスはそれぞれ独立したエンティティです。無条件に変数を共有できるわけではありません。

プロセスに類似する概念として、**スレッド**と呼ばれるものがあります。スレッドは、Microsoft Windows NTやWindows 95/98などのシステムで利用でき、ある1つのプロセス内部での独立した実行の流れを意味するものです。スレッドが利用可能な場合、コンピュータシステムのアーキテクチャによる制限はありますが、1つのプログラムの2つの部分を同時に実行することができます。たとえば、ワードプロセッサプログラムでは、文書をレイアウトしながらスペルチェックを行うことができます。この場合、同じプロセス内で2つ以上のスレッドが実行されており、各スレッドでは同じ変数とデータを対象に処理を行っています。

スレッドプログラミングにはそれなりに難しい側面がありますが、マルチスレッドデータベースなど、アプリケーションによっては非常に有益です。スレッドの主な利点は、オペレーティングシステムのタスクのような“重量級”プロセスではなく、“軽量”プロセスであるという点です。軽量プロセスは、オペレーティングシステムタスクのプロセスが行うアドレス空間の初期化やプロセスのスタートアップといったコストを伴いません。生存期間の短いプロセスの場合、これは大幅な効率の向上につながります。逆に、欠点は信頼性です。いずれかのスレッド（軽量プロセ

ス）がほかのすべてのスレッドをクラッシュさせる可能性があるからです。

現在のところ、UNIXによるスレッドベースの実行のサポートは限定的です。POSIX.4仕様では、UNIXのリアルタイムアプリケーションにとって有益なスレッド機構を定義しており、Linuxをはじめ、UNIX系システムへの実装が進められている段階です。本書では、現時点でのサポートが限定的であることから、スレッドについてはこれ以上取り上げません。ただし、今後急速に状況が変わる可能性もあります。スレッドベースのプログラミングに関心がある場合には、インターネット上の関連サイトなどで情報を収集してください。

10.5 シグナルとは

シグナルとは、一定の条件に応じてUNIXシステムによって生成されるイベントのことです。イベントを受け取ったプロセスはなんらかのアクションを起こすことがあります。シグナルは、メモリのセグメント違反、浮動小数点プロセッサのエラー、不正なインストラクションなど、エラーが発生した場合に生成されます。また、シェルと端末ハンドラによって割り込みを発生させるために生成されることもあります。プロセス間で情報をやりとりしたり、動作を変更したりするための方法として、あるプロセスから別のプロセスに明示的にシグナルが送られることもあります。いずれの場合もプログラミングインタフェースは同じであり、シグナルの生成、捕捉と対処、または（少なくともいくつかのシグナルでは）無視が可能です。

シグナルの名前は、ヘッダーファイル`signal.h`で定義されています。名前はいずれも`SIG`で始まり、次のようなものがあります。

表10.2 シグナルの種類（その1）

シグナル名	説明
SIGABORT	プロセスアボート*
SIGALRM	アラームクロック
SIGFPE	浮動小数点例外*
SIGHUP	ハングアップ
SIGILL	不正なインストラクション*
SIGINT	端末割り込み
SIGKILL	終了（kill。捕捉や無視はできない）
SIGPIPE	読み手のいないパイプへの書き込み
SIGQUIT	端末クイット
SIGSEGV	不正なメモリセグメントアクセス*
SIGTERM	終了
SIGUSR1	ユーザー定義シグナル1
SIGUSR2	ユーザー定義シグナル2

あらかじめシグナルを捕捉する設定を行っていない場合、これらのシグナルのいずれかを受け取ると、そのプロセスはただちに終了させられます。表中の*の付いたシグナルでは、システムに依存した処理が行われることもあります。通常、プロセスの終了時にはコアダンプファイルが作成されます。コアダンプファイルとは、現在のディレクトリにcoreという名前で作成されるファイルのことです。coreにはプロセスのイメージが保存されているので、デバッグを行う際に役立ちます。

上の表のほかにも次のようなシグナルがあります。

表10.3 シグナルの種類（その2）

シグナル名	説明
SIGCHLD	子プロセスが停止または終了した
SIGCONT	停止していた場合には実行を続行する
SIGSTOP	実行を停止する（捕捉や無視はできない）
SIGTSTP	端末の休止（stop）シグナル
SIGTTIN	バックグラウンドプロセスからの読み取りの試み
SIGTTOU	バックグラウンドプロセスからの書き込みの試み

SIGCHLDは、子プロセスの管理に利用することができます。このシグナルはデフォルトでは無視されます。これ以外のシグナルの場合、シグナルを受け取ったプロセスは停止します。ただし、SIGCONTだけは例外で、このシグナルを受け取ったプロセスは実行を再開します。これらのシグナルは、ジョブ制御のためにシェルプログラムによって使われ、ユーザープログラムで使われるケースはまれです。

最初の表にあげたシグナルについては、後でもう少し詳しく取り上げますが、ここではSIGINTを例にシグナル処理について説明しておきます。シェルと端末ドライバが適切に設定されている場合、キーボードから割り込み文字（一般に**Ctrl - C**）を入力すると、SIGINTシグナルが生成され、このシグナルがフォアグラウンドプロセス（現在実行されているプロセス）に送信されます。シグナルを捕捉するように設定していないプログラムがこのシグナルを受け取ると、そのプログラムは終了させられますが、signalライブラリ関数を使えば、シグナルを処理することができます。

```
#include <signal.h>

void (*signal(int sig, void (*func)(int)))(int);
```

わかりにくい宣言ですが、signalが2つのパラメータsigとfuncを取る関数として宣言されています。パラメータsigには、捕捉または無視するシグナルを指定します。funcには、指定したシグナルを受け取ったときに呼び出す関数を指定します。この関数は、int型の引数（受け取ったシグナル）を1つだけ取るvoid型の関数でなければなりません。signal関数自体は、同じ型の関数（該当するシグナルを処理するために以前にセットアップした関数）を返すか、また

は次のいずれかの特別な値を返します。

- `SIG_IGN`　　シグナルを無視する。
- `SIG_DFL`　　デフォルトの動作に戻す。

Linuxシステムでは、指定されたシグナルが捕捉された後、自動的にデフォルトの動作に戻されます。上の値以外にSIG_ERRという特別な値もありますが、これはエラーコードとして返される値であり、signalの呼び出しで使用することはできません。

さっそくサンプルプログラムを作成してシグナルの動作を確かめてみましょう。サンプルプログラムctrlc.cは、Ctrl-Cが入力されたときに、プログラムを終了せずに適切なメッセージを表示します。もう一度Ctrl-Cが入力されると、プログラムは終了します。

例題 シグナル処理

1 signal関数のパラメータsigに指定するシグナル処理関数ouchを用意します。

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

void ouch(int sig)
{
    printf("OUCH! - I got signal %d\n", sig);
}
```

2 main関数では、Ctrl-Cが押されたときに生成されるSIGINTシグナルを捕捉する関数をセットアップします。その後の無限ループでは、1秒おきにメッセージを表示します。

```
int main()
{
    (void) signal(SIGINT, ouch);

    while(1) {
        printf("Hello World!\n");
        sleep(1);
    }
}
```

3 プログラムの実行中にCtrl-Cを入力すると、プログラムからメッセージが表示され、実行は継続されます。もう一度Ctrl-Cを入力すると、プログラムは終了します。

```
$ ctrlc
Hello World!
Hello World!
```

```
Hello World!
Hello World!
^C
OUCH! - I got signal 2
Hello World!
Hello World!
Hello World!
Hello World!
^C
$
```

このサンプルプログラムでわかるように、シグナル処理関数は**int**型の引数を1つだけ取ります。この引数は、関数の呼び出しの原因となったシグナル番号です。このことを利用すれば、1つの関数で複数のシグナルを処理することができます。サンプルプログラムでは、単純に**SIGINT**の値を出力しています。このシステムでは値は2ですが、シグナルを表すのに数値を使うのは避けたほうが無難です。自分で作成するプログラムでは、数値ではなくシグナル名を使ってください。

解説

このプログラムでは、**Ctrl - C**の入力によって**SIGINT**シグナルが生成されたときに呼び出される関数**ouch**を設定しています。関数**ouch**を抜けると、プログラムの実行は続けられますが、シグナル処理は自動的にデフォルトの動作に戻されます。したがって、次に**SIGINT**シグナルを受けとると、プログラムは終了します（デフォルトの動作）。



注意

> Berkely UNIXから派生したバージョンなど、一部のUNIXのバージョンでは、シグナル処理をデフォルトの動作に戻しません。したがって、このようなバージョンのUNIXでは、明示的に**signal**を呼び出してデフォルトの動作に戻す必要があります。具体的には、関数**ouch**の中に次の行を追加します。
>
> ```
> signal(SIGINT, SIG_DFL);
> ```

シグナルハンドラをそのままにして、引き続き**Ctrl - C**を処理したい場合には、もう一度**signal**関数を呼び出してシグナル処理を設定する必要があります。ただし、この場合、呼び出された関数の開始からシグナルハンドラが再設定されるまでの短い間、シグナルが処理されないことになり、この間に受け取ったシグナルによってプログラムが終了してしまう可能性があります。こうした問題を避けるには、**signal**関数を使わず、後で説明する別のインタフェースを使います。

signal関数は、指定したシグナルのシグナルハンドラが存在していた場合にはその値を返します。エラーが発生した場合には**SIG_ERR**を返し、**errno**に正の値が設定されます。無効なシグナルが指定されたり、捕捉や無視のできない**SIGKILL**などのシグナルを処理する試みが行われた場合には、**errno**に**EINVAL**が設定されます。

10.6 シグナルの送信

プロセスが自分自身にシグナルを送信するには、**raise**を使います。

```
#include <signal.h>

int raise(int sig);
```

指定されたシグナルに対するシグナルハンドラがある場合には、そのシグナルハンドラが実行されます。**raise**は、成功すると0を返します。引数が有効なシグナル番号でない場合には0以外の値を返し、**errno**には**EINVAL**が設定されます。

プロセスから別のプロセス（自分自身を含む）にシグナルを送信するには、**kill**を使います。この関数は、同名のシェルコマンドと等価です。

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
```

kill関数は、**pid**で指定されたプロセスIDを持つプロセスに対して、**sig**に指定されたシグナルを送信します。成功すると0を返します。シグナルを送信するには、送信側のプロセスがシグナルを送信するためのパーミッションを持っていなければなりません。これは通常、どちらのプロセスも同じユーザーIDを持っていなければならないことを意味します。つまり、シグナルの送信先として指定できるのは、自分が実行しているプロセスのうちのいずれかです。ただし、スーパーユーザーは任意のプロセスにシグナルを送ることができます。

killは失敗すると -1を返し、**errno**にはエラーの内容を示す値が設定されます。**errno**に設定される値は、指定されたシグナルが有効でない場合には**EINVAL**、パーミッションがない場合には**EPERM**、指定されたプロセスが存在しない場合には**ESRCH**です。

シグナルにはアラーム時計の機能もあります。**alarm**関数を使うと、一定時間後に**SIGALRM**シグナルを生成することができます。

```
#include <unistd.h>

unsigned int alarm(unsigned int seconds);
```

alarmは、**seconds**に指定された秒数の経過後に**SIGALRM**シグナルをプロセスに配送するようにスケジュールします。実際には、処理のための時間とスケジューリングの遅延から、指定された時間よりわずかに遅れてシグナルが配送されます。引数に0を指定した場合には、すでに登録されているアラーム要求がキャンセルされます。シグナルを受け取る前に**alarm**を呼び出した場合には、アラームが再スケジューリングされます。各プロセスが登録できるアラームは1つだけです。

alarmは、登録済みのアラームが送信されるまでの残り秒数を返します。

alarmの動作を理解するために、forkとsleep、signalを使って、alarmの動作をシミュレートするプログラムを作成してみましょう。プログラムの名前はalarm.cとします。

例題 アラーム時計

1 最初のdingは、アラーム時計をシミュレートする関数です。

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

void ding(int sig)
{
    printf("alarm has gone off\n");
}
```

2 main関数では子プロセスを作成します。子プロセスは、5秒経過すると親プロセスにSIGALRMを送信します。

```
int main()
{
    int pid;

    printf("alarm application starting\n");

    if((pid = fork()) == 0) {
        sleep(5);
        kill(getppid(), SIGALRM);
        exit(0);
    }
```

3 親プロセスでは、SIGALRMを捕捉するためにsignalを呼び出し、目的のシグナルを待ちます。

```
    printf("waiting for alarm to go off\n");
    (void) signal(SIGALRM, ding);

    pause();

    printf("done\n");
    exit(0);
}
```

実行すると、シグナルを受け取るまでの5秒間、プログラムは停止します。

```
$ alarm
alarm application starting
waiting for alarm to go off
<5秒間停止>
alarm has gone off
done
$
```

このプログラムでは、新しい関数**pause**を使っています。**pause**を呼び出すと、シグナルを受け取るまでの間、プログラムは実行を一時停止します。シグナルを受け取ると、設定されたハンドラが実行され、プログラムは通常どおり実行を続けます。

```
#include <unistd.h>

int pause(void);
```

pause関数は、(次に受け取ったシグナルがプログラムを終了させない場合には) -1を返し、シグナルによって割り込まれた場合には**errno**に**EINTR**を設定します。

解説

このアラーム時計シミュレーションプログラムでは、**fork**を使って新しいプロセスを開始します。子プロセスは5秒間スリープした後、親プロセスに**SIGALRM**を送信します。親プロセスの側では**SIGALRM**を捕捉するように設定した後、シグナルを受け取るまで実行を中断します。

シグナルの使用、および実行の一時停止は、UNIXプログラミングの重要な要素のひとつです。UNIXでは、必ずしもプログラムをずっと実行している必要はありません。絶えずループを回りながらイベントが発生したかどうかを調べなくてもイベントの発生を待つことができます。1つのプロセッサを複数のプロセスが共有し、ビジーウェイトがシステムパフォーマンスに大きく影響するマルチユーザー環境では、この点は特に重要な意味を持っています。

注意

> シグナルを使うプログラムでは、いわゆるレースコンディションが起きる可能性があるので、慎重なプログラミングが要求されます。たとえば、あるシグナルを待つ目的で**pause**を呼び出している場合、目的のシグナルが**pause**の呼び出しよりも前に発生すると、呼び出し側のプログラムは、発生することのないシグナルを永久に待つことになります。この種のレースコンディションやクリティカルなタイミングの問題は、プログラミング初心者のおちいりやすいわなです。シグナルを扱うプログラムを作成する場合には、常に細心の注意をはらってコードをチェックする必要があります。

10.6.1 信頼性の高いシグナルインタフェース

これまでは、**signal**とその関連関数を使ったシグナルの送信と捕捉について取り上げました。これらの関数は実際のUNIXプログラムではよく使われています。ただし、X/Open仕様では、シグナルを扱うためのより信頼性の高い新しいプログラミングインタフェースとして**sigaction**の使用を推奨しています。

```
#include <signal.h>

int sigaction(int sig, const struct sigaction *act, struct sigaction *oact);
```

sigaction構造体は、**sig**で指定されたシグナルを受け取ったときの動作を設定するのに使われる構造体で、**signal.h**で定義されています。**sigaction**構造体には、少なくとも次のメンバがあります。

void (*) (int) sa_handler	関数、**SIG_DFL**または**SIG_IGN**
sigset_t sa_mask	**sa_handler**の実行中にブロックされるシグナル
int sa_flags	シグナルの動作を変更するためのフラグ

sigaction関数は、シグナル**sig**に関連付けられた動作を設定します。パラメータ**oact**が**NULL**でなければ、**oact**で示される場所に以前のシグナルの動作が書き込まれます。パラメータ**act**が**NULL**の場合、**sigaction**関数が行う処理はこれだけです。パラメータ**act**が**NULL**でない場合、**sigaction**関数は指定されたシグナルに対する動作を設定します。

signal関数同様、**sigaction**関数は成功すると0を返し、それ以外の場合には -1を返します。指定されたシグナルが無効な場合や、捕捉や無視のできない**SIGKILL**などのシグナルを処理する試みが行われた場合、**errno**には**EINVAL**が設定されます。

引数**act**で示される**sigaction**構造体のうち、メンバ**sa_handler**は、シグナル**sig**を受け取ったときに呼び出される関数へのポインタです。この関数は、**signal**関数に渡される関数**func**と同じような働きをします。**sa_handler**には、特別な値として**SIG_IGN**（シグナルを無視）または**SIG_DFL**（デフォルトの動作に戻す）を指定することができます。

メンバ**sa_mask**には、**sa_handler**関数を呼び出す前にプロセスのシグナルマスクに追加するシグナルの集合を指定します。これらのシグナルはブロックされ、プロセスには配送されません。したがって、前に指摘した、シグナルハンドラが実行を終える前にシグナルを受け取った場合の問題も回避することができます。

ただし、**sigaction**で設定されたハンドラによって捕捉されるシグナルの動作は、デフォルトではリセットされません。このため、**signal**関数の場合と同様にデフォルトの動作に戻したければ、**SA_RESETHAND**を含む値をメンバ**sa_flags**に設定しなければなりません。

sigactionについてさらに詳しく見る前に、サンプルプログラム**ctrlc.c**を**sigaction**を使って書き換えてみましょう。

例題 sigaction

ctrlc.cを次のように修正し、**SIGINT**を**sigaction**で処理するようにします。修正後のプログラムは**ctrlc2.c**とします。

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

void ouch(int sig)
{
    printf("OUCH! - I got signal %d\n", sig);
}

int main()
{
    struct sigaction act;

    act.sa_handler = ouch;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;

    sigaction(SIGINT, &act, 0);

  while(1) {
    printf("Hello World!\n");
    sleep(1);
  }
}
```


プログラムを実行し、**Ctrl-C**を入力すると、常に同じメッセージが表示されます。これは、**SIGINT**が繰り返し**sigaction**によって処理されているためです。プログラムを終了するには**Ctrl-**を入力します。このキーは、デフォルトで**SIGQUIT**シグナルを生成します。

```
$ ctrlc2
Hello World!
Hello World!
Hello World!
^C
OUCH! - I got signal 2
Hello World!
Hello World!
^C
OUCH! - I got signal 2
Hello World!
Hello World!
^\
Quit
$
```

解説

このプログラムでは、**signal**の代わりに**sigaction**を使って、**Ctrl-C**（**SIGINT**）のシグナルハンドラに関数**ouch**を設定しています。まず、目的のハンドラ、シグナルマスク、フラグを含む**sigaction**構造体をセットアップします。このプログラムの場合、特にフラグを指定する必要はなく、**sigemptyset**関数を使って空のシグナルマスクを作成しています。

10.6.2 シグナルの集合

ヘッダーファイル**signal.h**では、シグナルの集合を操作するための**sigset_t**型と各種関数が定義されています。シグナルの集合は、シグナルを受け取ったときのプロセスの動作を変更する**sigaction**やその他の関数で使われます。

```
#include <signal.h>

int sigaddset(sigset_t *set, int signo);
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigdelset(sigset_t *set, int signo);
```

上の関数は、それぞれの名前が示す処理を行います。**sigemptyset**はシグナルの集合を空に初期化します。**sigfillset**は、定義済みのすべてのシグナルを含むようにシグナルの集合を初期化します。**sigaddset**と**sigdelset**はシグナルの集合に対して、指定されたシグナル（**signo**）を追加または削除します。これらの関数は、いずれも成功すると0を返します。失敗した場合には -1を返し、**errno**にエラーの内容を示す値を設定します。エラーとして定義されているのは、指定されたシグナルが無効であることを示す**EINVAL**だけです。

指定したシグナルがシグナルの集合のメンバかどうかを調べる関数として**sigismember**があります。**sigismember**関数は、指定したシグナルがシグナルの集合のメンバである場合には1を返し、メンバでない場合には0を返します。また、シグナルが無効な場合には -1を返し、**errno**に**EINVAL**を設定します。

```
#include <signal.h>

int sigismember(sigset_t *set, int signo);
```

プロセスのシグナルマスクを設定したり調べたりするには、**sigprocmask**関数を使います。プロセスのシグナルマスクとは、現在ブロックされているためにプロセスが受け取らないシグナルの集合のことです。

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
```

sigprocmaskでは、引数**how**によってさまざまな方法でプロセスのシグナルマスクを変更することができます。引数**set**が**NULL**でなければ、シグナルマスクの新しい値が**set**によって渡されます。以前のシグナルマスクは、シグナルの集合**oset**に書き込まれます。

引数**how**には次のいずれかを指定できます。

- **SIG_BLOCK** **set**に含まれるシグナルをシグナルマスクに追加する。
- **SIG_SETMASK** **set**から現在のシグナルマスクを設定する。
- **SIG_UNBLOCK** **set**に含まれるシグナルをシグナルマスクから削除する。

引数**set**がヌルポインタの場合、**how**の値は使われず、現在のシグナルマスクの値を取得するために**sigprocmask**を呼び出すことになります（現在のシグナルマスクの値は**oset**に書き込まれます）。

sigprocmask関数は、成功すると0を返します。パラメータ**how**が無効な場合には-1を返し、**errno**には**EINVAL**が設定されます。

シグナルがプロセスによってブロックされている場合、そのシグナルは配送されず、保留中（pending）になっています。**sigpending**を呼び出すと、ブロックされているシグナルのうち保留されているシグナルはどれかを調べることができます。

```
#include <sigpending>

int sigpending(sigset_t *set);
```

sigpendingは、配送されずにブロックされているシグナルのうち保留中のシグナルの集合を、**set**で示されるシグナルの集合に書き込みます。**sigpending**は成功すると0を返します。それ以外の場合には-1を返し、**errno**にエラーの内容を示す値を設定します。**sigpending**は、プログラムでシグナルを処理する必要があり、シグナル処理関数を呼び出す時期を制御したい場合に便利な関数です。

sigsuspendを呼び出すと、シグナルの集合のうちいずれかの配送があるまで、プロセスの実行を一時停止することができます。これは、すでに説明した**pause**よりも汎用性を高めた関数です。

```
#include <signal.h>

int sigsuspend(const sigset_t *sigmask);
```

sigsuspend関数は、**sigmask**に指定されたシグナルの集合でプロセスのシグナルマスクを置き換えたあと、プロセスの実行を一時停止します。受け取ったシグナルがプログラムを終了させ

る場合には、sigsuspendから戻ることはありません。受け取ったシグナルがプログラムを終了させない場合、sigsuspendは-1を返し、errnoにはEINTRが設定されます。

10.6.3 sigactionのフラグ

sigaction関数で使われる構造体sigactionのメンバsa_flagsでは、シグナルの動作を変更するために次のような値を指定できます。

- **SA_NOCLDSTOP** 子プロセスが停止したときにSIGCHLDを生成しない。
- **SA_RESETHAND** シグナルを受け取った時点でシグナルの動作をSIG_DFLにリセットする。
- **SA_RESTART** EINTRによるエラーにせずに割り込み可能関数を再起動する。
- **SA_NODEFER** 捕捉時にシグナルをシグナルマスクに追加しない。

SA_RESETHANDフラグを使うと、シグナルを捕捉したときに自動的にシグナル関数をクリアし、デフォルトの動作に戻すことができます。

プログラムで使われる多くのシステムコールは割り込み可能であり、シグナルを受け取るとエラーを返します。このとき、errnoには、シグナルを受け取って関数が戻ったことを示すEINTRが設定されます。シグナルを使ったアプリケーションでは、こうした動作について十分に注意する必要があります。sigaction関数を呼び出すときにsa_flagsにSA_RESTARTが設定されていると、この設定がなければシグナルによって割り込まれていた関数が、シグナル処理関数の実行後に再起動されることになります。

通常、シグナル処理関数が実行されている場合には、受け取ったシグナルが、そのシグナル処理関数が実行されている間、プロセスのシグナルマスクに追加されます。このことによって同じシグナルが続けて生起するのを防ぎますが、シグナル処理関数はもう一度実行されることになり、シグナル処理関数が再入可能（リエントラント）でない場合には問題が起きる可能性があります。ただし、SA_NODEFERフラグが設定されている場合には、シグナルを受け取ってもシグナルマスクは変更されません。

シグナル処理関数は、途中で割り込まれてほかから再度呼び出される可能性があります。この場合、最初の呼び出しに戻ったときに適切に動作できるようになっていなければなりません。これは、（自分自身を呼び出す再帰ではなく）再入が可能（もう一度入って問題なく実行できること）でなければならないことを意味しています。同時に複数のデバイスを処理する割り込みサービスルーチンは、同じコードの実行中により優先度の高い割り込みがかかる可能性があるので、再入可能である必要があります。

再入可能であること、あるいはシグナルを生起しないことがX/Open仕様によって保証されていて、シグナルハンドラの中から安全に呼び出すことのできる関数には次のものがあります。

access	fstat	read	sysconf
alarm	getegid	rename	tcdrain
cfgetispeed	geteuid	rmdir	tcflow
cfgetospeed	getgid	setgid	tcflush
cfsetispeed	getgroups	setpgid	tcgetattr
cfsetospeed	getpgrp	setsid	tcgetpgrp
chdir	getpid	setuid	tcsendbreak
chmod	getppid	sigaction	tcsetattr
chown	getuid	sigaddset	tcsetpgrp
close	kill	sigdelset	time
creat	link	sigemptyset	times
dup2	lseek	sigfillset	umask
dup	mkdir	sigismember	uname
execle	mkfifo	signal	unlink
execve	open	sigpending	utime
_exit	pathconf	sigprocmask	wait
fcntl	pause	sigsuspend	waitpid
fork	pipe	sleep	write
stat			

10.7 よく使われるシグナルのリファレンス

次に示すのは、UNIXプログラムでよく使われるシグナルと、そのデフォルトの動作の説明です。

表10.4 よく使われるシグナル（その1）

シグナル名	説明
SIGALRM	alarm関数が設定するタイマによって生成される。
SIGHUP	切断する端末から制御プロセスへ、また終了時に制御プロセスから各フォアグラウンドプロセスに送られる。
SIGINT	通常、**Ctrl-C**またはその他の設定された割り込み文字を入力したときに端末から生成される。
SIGKILL	このシグナルは捕捉や無視ができないので、主に不要なプロセスを強制的に終了させるためにシェルで使われる。
SIGPIPE	読み手のいないパイプへの書き込みが行われた場合に生成される。
SIGTERM	プロセス終了の要求として送られ、UNIXではシャットダウン時にシステムサービスを停止させるために使われる。このシグナルは、killコマンドによって送信されるデフォルトのシグナルである。
SIGUSR1 SIGUSR2	ステータス情報の報告を要求するなど、複数のプロセスが互いに通信する場合に使用できる。

上の表にあげたシグナルについては、デフォルトの動作はプロセスの異常終了です。この場合、結果は`_exit`（`exit`に似ていますが、カーネルに戻る前にクリーンアップを行いません）が呼び出された場合と同じですが、`wait`と`waitpid`でステータスを取得でき、シグナルによる異常終了であることがわかります。

表10.5 よく使われるシグナル（その2）

シグナル名	説明
SIGFPE	浮動小数点演算例外によって生成される。
SIGILL	プロセッサによって不正なインストラクションが実行されたときに生成される。一般に、破壊されたプログラムや無効な共有メモリモジュールが原因。
SIGQUIT	通常、**Ctrl**-\またはその他の設定されたQUIT文字を入力したときに端末から生成される。
SIGSEGV	セグメンテーション違反。通常、配列境界を超えたり、無効なポインタでメモリを参照するなど、メモリ内の不正な場所への読み取りや書き込みを行ったときに生成される。ローカルな配列変数の操作を誤ってスタックを破壊すると、関数が不正なアドレスに戻ることになり、このとき`SIGSEGV`が生成されることがある。

この表のシグナルについても、デフォルトの動作は異常終了です。さらに、コアファイルの作成など、システムに依存する処理が行われる場合もあります。

表10.6 よく使われるシグナル（その3）

シグナル名	説明
SIGSTOP	実行を停止する（捕捉や無視はできない）。
SIGTSTP	端末の休止（stop）シグナル。通常、**Ctrl**-**Z**を入力したときに生成される。
SIGTTIN	シェルに対して、端末からの読み取りや出力の生成が必要なためにバックグラウンドジョブが停止されていることを伝えるのに使われる。

デフォルトでは、上の表のいずれかのシグナルを受け取ったプロセスは停止します。

表10.7 よく使われるシグナル（その4）

シグナル名	説明
SIGCONT	停止していた場合には実行を続行する。

`SIGCONT`は、停止していたプロセスを再開します。プロセスが停止していなかった場合には無視されます。

表10.8 よく使われるシグナル（その5）

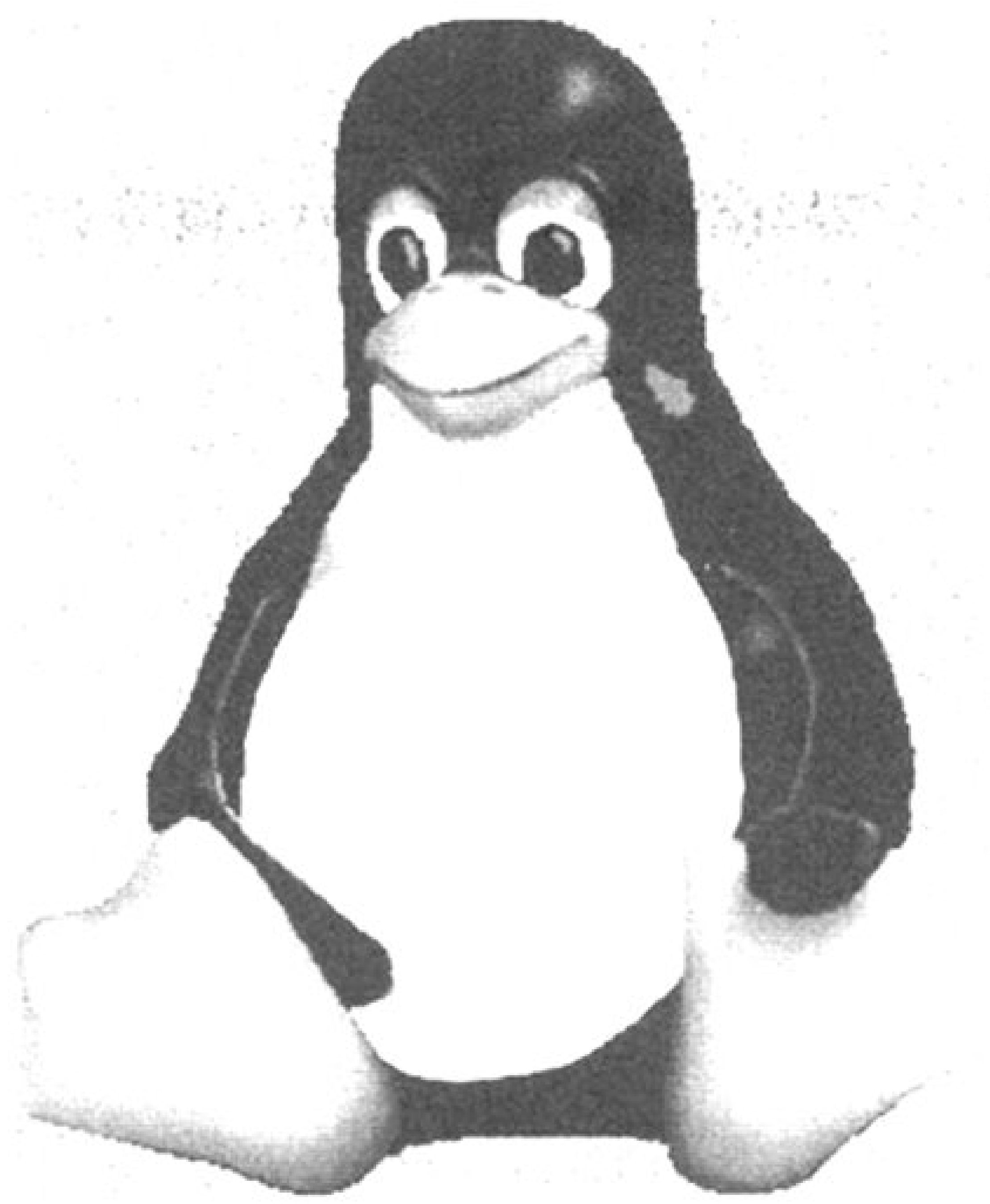
シグナル名	説明
SIGCHLD	子プロセスが停止または終了したときに生成される。

デフォルトでは、`SIGCHLD`は無視されます。

10.8 この章のまとめ

この章では、UNIXオペレーティングシステムの重要な要素であるプロセスについて取り上げ、プロセスの起動、終了、表示の方法、およびプログラムでのプロセスの使い方について学びました。

また、実行中のプログラムの動作を制御するために使用できるイベントとして、シグナルについても取り上げました。



第11章
プロセス間通信とパイプ

第10章では、プロセス間でメッセージをやりとりする方法として、シグナルを使って別のプロセスにメッセージを送る方法を取り上げました。これは方法としてはごく単純なもので、割り込みによって応答を引き出すことはできますが、転送できる情報はシグナル番号だけに限られます。

この章では、プロセス間で情報を交換するためのより便利な方法としてパイプを取り上げます。この章の最後では、新しく学んだ知識を応用してCDデータベースプログラムをクライアントサーバーアプリケーションとして実装し直します。取り上げる項目は次のとおりです。

- パイプの定義
- プロセスのパイプ
- **pipe**関数
- 親プロセスと子プロセス
- 名前付きパイプ(FIFO)
- クライアントサーバーアプリケーション設計のポイント

11.1 パイプとは

パイプとは、あるプロセスからのデータフローを別のプロセスにつなぐことです。一般には、あるプロセスの出力を別のプロセスの入力に接続します。

UNIXでは、しばしば複数のシェルコマンドをつなげて、あるプロセスからの出力を別のプロセスの入力として利用します。たとえば、次のような形式のコマンドは非常によく使われます。

```
cmd1 | cmd2
```

上のようなコマンドが入力されると、シェルは2つのコマンド**cmd1**と**cmd2**の標準入力と標準出力を次のように結び付けます。

- **cmd1**への標準入力を端末キーボードから得る。
- **cmd1**からの標準出力を**cmd2**の標準入力に渡す。
- **cmd2**からの標準出力を端末画面に接続する。

つまり、シェルは標準入出力ストリームをつなぎ直し、キーボードから入力されたデータが2つのコマンドを経由して画面に出力されるようにします。

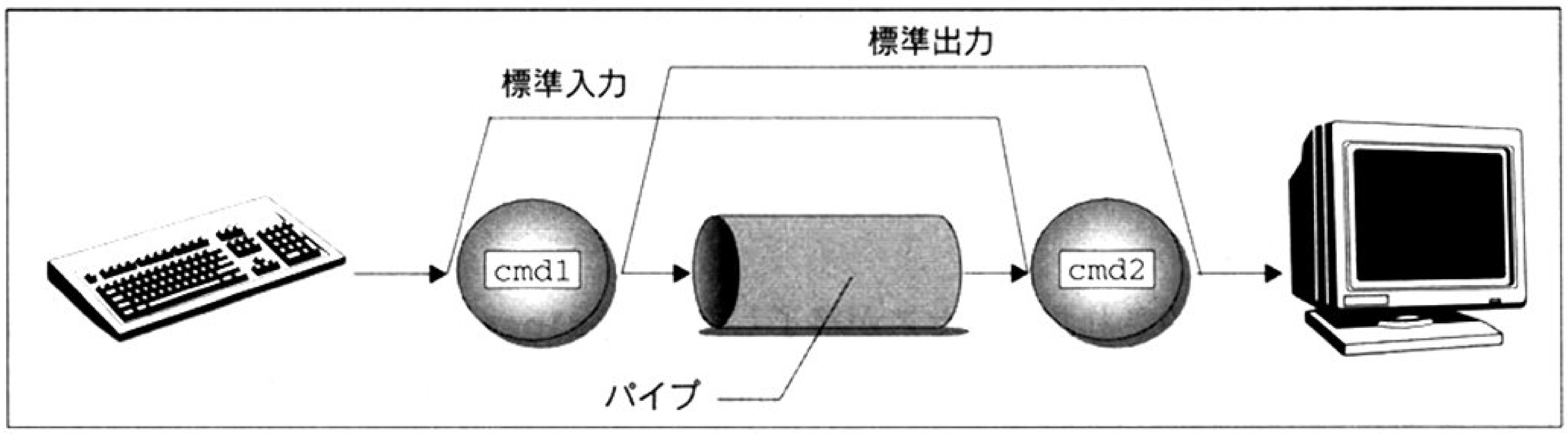


図11.1 パイプの概念

この章では、このようなパイプのしくみをプログラムの中で実現する方法を示します。また、パイプを使って複数のプロセスを接続し、簡単なクライアントサーバーシステムを作成します。

11.2 プロセスパイプ

2つのプログラム間でデータを受け渡しするための最も簡単な方法は、**popen**関数と**pclose**関数を使うことです。

```
#include <stdio.h>

FILE *popen(const char *command, const char *open_mode);
int pclose(FILE *stream_to_close);
```


◆ popen

popen関数を使うと、別のプログラムを新しいプロセスとして呼び出し、このプロセスとの間でデータの受け渡しを行うことができます。パラメータ**command**には、実行するプログラムの名前とプログラムに渡す引数を指定します。**open_mode**は**r**または**w**のいずれかでなければなりません。

open_modeが**r**の場合には、呼び出されたプログラムの出力が呼び出し側のプログラムで利用可能になります。この出力は、**popen**によって返されたファイルストリームから読み取ることができます。読み取りには、**fread**などの通常の標準入出力ライブラリ関数を使うことができます。**open_mode**が**w**の場合には、**fwrite**などを使って、呼び出された側のコマンドにデータを送ることができます。通常、呼び出された側のプログラムでは、ほかのプロセスからのデータを読み取っているかどうかは関知せず、単に標準入出力を読み取って処理を加えます。

popenを呼び出すときには、必ず**r**または**w**のどちらかを指定する必要があります（**popen**の標準の実装では、これ以外のオプションはサポートされていません）。換言すれば、**popen**では

読み書き用にほかのプログラムを呼び出すことはできません。popenは、失敗した場合にはNULLを返します。

◆ pclose

popenで開始されたプロセスが終了したら、pcloseを呼び出し、関連付けられているファイルストリームをクローズします。pcloseの呼び出しは、popenで開始されたプロセスが終了した時点で戻ります。pcloseを呼び出したときにプロセスがまだ実行中だった場合、pcloseはプロセスが終了するのを待ちます。

通常、pcloseは、クローズするファイルストリームに関連付けれらたプロセスの終了コードを返します。呼び出し側のプロセスが、pcloseを呼び出す前にwaitを実行していた場合、終了コードは失われ、pcloseは-1を返します。このとき、errnoにはECHILDが設定されます。

さっそくpopenとpcloseを使って簡単なプログラムを作成してみましょう。名前はpopen1.cとします。このプログラムでは、popenを使って、unameから出力される情報を取得します。uname -aコマンドは、マシンの種類、OS名、バージョン、リリース、マシンのホスト名など、システム情報を表示します。

例題 popenとpcloseの使用

必要な初期化を行ったあと、unameコマンドを指定してパイプをオープンし、read_fpで出力を読み取れるようにします。処理を終えたら、read_fpが示すパイプをクローズします。

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main()
{
    FILE *read_fp;
    char buffer[BUFSIZ + 1];
    int chars_read;
    memset(buffer, '\0', sizeof(buffer));
    read_fp = popen("uname -a", "r");
    if (read_fp != NULL) {
        chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
        if (chars_read > 0) {
            printf("Output was:-\n%s\n", buffer);
        }
        pclose(read_fp);
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

次に、筆者の使っているマシンでの出力例を示します。

```
$ popen1
Output was:-
Linux euripide 2.0.35 #1 Tue Oct 13 04:01:38 JST 1998 i586 unknown
```

解説

popenを使用し、-aオプションを指定してunameコマンドを呼び出しています。呼び出しが成功した場合には、返されたファイルストリームを使ってBUFSIZ文字までのデータを読み取り、画面に表示します（BUFSIZはstdio.hで定義されています）。このプログラムでは、unameからの出力をプログラムの中に取り込んでいるので、必要に応じて加工することもできます。

11.2.1 popenへの出力の送信

popen1.cは外部プログラムからの出力を受け取るサンプルプログラムでした。今度は、外部プログラムに出力を送るpopen2.cを作成してみましょう。外部プログラムにはodコマンドを使います。

例題 外部プログラムへの出力の送信

popen2.cのコードは次のとおりです。

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main()
{
    FILE *write_fp;
    char buffer[BUFSIZ + 1];

    sprintf(buffer, "Once upon a time, there was...\n");
    write_fp = popen("od -c", "w");
    if (write_fp != NULL) {
        fwrite(buffer, sizeof(char), strlen(buffer), write_fp);
        fclose(write_fp);
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

プログラムを実行すると、次のように出力されます。

```
$ popen2
0000000   O   n   c   e       u   p   o   n       a       t   i   m   e
0000020   ,       t   h   e   r   e       w   a   s   .   .   .  \n
0000037
```

解説

このプログラムでは、パラメータwを指定してpopenを呼び出し、od -cコマンドを起動しています。次に、あらかじめ用意した文字列をod -cコマンドに送ります。od -cコマンドは、送られた文字列を処理して結果を標準出力に表示します。

コマンドラインで次のように実行すれば、同じ結果を得ることができます。

```
$ echo "Once upon a time, there was..." | od -c
```

11.2.2 大量のデータのやりとり

これまでの例では、freadまたはfwriteを1回だけ呼び出してすべてのデータをやりとりしていました。しかし、場合によってはデータをいくつかに分けたり、サイズのわからないデータを処理したいこともあります。このような場合には、大きなバッファを用意するのではなく、freadやfwriteを複数回呼び出せば、データを小分けにして処理することができます。

次のサンプルプログラムpopen3.cでは、パイプのデータをすべて読み取ります。

例題 パイプからの大量のデータの読み取り

このプログラムでは、ps -axの出力を読み取ります。プログラムを実行するまでは出力がどれくらいになるかわからないので、パイプの読み取りを何度か実行する必要があります。

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main()
{
    FILE *read_fp;
    char buffer[BUFSIZ + 1];
    int chars_read;

    memset(buffer, '\0', sizeof(buffer));
    read_fp = popen("ps -ax", "r");
    if (read_fp != NULL) {
```

```
        chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
        while (chars_read > 0) {
            printf("Reading:-\n %s\n", buffer);
            chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
        }
        pclose(read_fp);
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

プログラムを実行したときの出力例を示します（一部省略しています）。

```
$ popen3
Reading:-
   PID TTY STAT  TIME COMMAND
     1  ?  S     5:39 init [3]
     2  ?  SW    0:27 (kflushd)
     3  ?  SW<   0:00 (kswapd)
...
    13  ?  S     1:33 /sbin/update
    81  ?  S     4:28 /usr/sbin/syslogd
    83  ?  S     0:00 /usr/sbin/klogd
   106  ?  S     0:01 /usr/sbin/crond -l10
Reading:-
23264  p5 S     0:00 popen3
23265  p5 R     0:00 ps -ax
...
```

解説

このプログラムでは、**popen1.c**と同様、パラメータ**r**を指定して**popen**を呼び出しています。ただし、今度は利用可能なデータがなくなるまでファイルストリームからの読み取りを続けます。**ps**コマンドの実行には一定の時間がかかりますが、**ps**コマンドと**popen3**プログラムのどちらも、実行可能なときに実行するようにスケジューリングされます。読み手側のプロセスである**popen3**に対する入力データがない場合、**popen3**は、入力データが利用可能になるまで一時停止します。書き手側のプロセスの**ps**がバッファに収まりきらない出力を生成した場合、書き手側のプロセスは、読み手側がデータを処理するまで一時停止します。

11.2.3 popenの実装

popenは要求されたプログラムを実行するときに、まずシェル**sh**を起動し、文字列**command**を引数として渡します。

UNIXでは、パラメータの展開はシェルによって行われます。このため、シェルを呼び出して

から指定したプログラムを実行するpopenでは、プログラムを起動する前に*.cといったワイルドカードをシェルに展開させることができます。このことを利用すれば、popenからかなり複雑なシェルコマンドを起動することもできます。これはpopenの利点です。プロセスを作成するときに使うその他の関数、たとえばexeclなどでは、プログラムを呼び出す前にシェル展開をプログラマが行う必要があり、呼び出し方が複雑になってしまいます。

ただし、シェルを使うことの副作用もあります。それは、popenを呼び出すたびに、要求されたプログラムだけでなくシェルも起動される点です。つまり、1回のpopenの呼び出しで2つのプロセスが起動されることになります。このため、popen関数は、システムリソースの点からいえばやや高価な関数です。

次に示すのは、上に述べたpopenの動作を確認するためのサンプルプログラムpopen4.cです。このプログラムでは、popen*というワイルドカードにマッチするサンプルプログラムのソースファイルをすべてcatし、これをパイプでwc -lに渡すことによって、全体の行数をカウントします。これは、コマンドラインで次のコマンドを実行することと同じです。

```
$ cat popen*.c | wc -l
```

例題 popenによるシェルの起動

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main()
{
    FILE *read_fp;
    char buffer[BUFSIZ + 1];
    int chars_read;

    memset(buffer, '\0', sizeof(buffer));
    read_fp = popen("cat popen*.c | wc -l", "r");
    if (read_fp != NULL) {
        chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
        while (chars_read > 0) {
            printf("Reading:-\n %s\n", buffer);
            chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
        }
        pclose(read_fp);
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

プログラムを実行すると、次のように出力されます。

```
$ popen4
Reading:-
      101
```

解説

このプログラムの実行結果から、**popen**によってシェルが呼び出され、プログラムの中で指定した**popen*.c**という文字列が、**popen**で始まり**.c**で終わるすべてのファイルのリストに展開されることがわかります。また、パイプ記号がシェルで実行した場合と同様に処理され、**cat**からの出力が**wc**に渡されています。つまり、このプログラムでは1回の**popen**の呼び出しで、シェル、**cat**プログラム、**wc**を呼び出して出力をリダイレクトしていることになります。実行結果では、最終出力だけが表示されています。

11.3 pipe関数

これまで見てきたのは高水準の**popen**関数でしたが、ここではより低水準の**pipe**関数について説明します。**pipe**関数を使うと、指定したコマンドの解釈に必要なシェル起動のオーバーヘッドなしに、2つのプログラム間でデータをやりとりすることができます。また、データの読み取りや書き込みをより細かく制御することもできます。

pipe関数のプロトタイプ宣言は次のとおりです。

```
#include <unistd.h>

int pipe(int file_descriptor[2]);
```

pipe関数は、2つのファイルデスクリプタを要素として持つ配列（へのポインタ）を引数に取ります。成功すると、新しい2つのファイルデスクリプタを配列にセットし、0を返します。失敗した場合には-1を返し、エラーの内容を示す値が**errno**に設定されます。Linuxのマニュアルページでは、次のエラーが定義されています。

- **EMFILE**　　プロセスによって使われているファイルデスクリプタが多すぎる
- **ENFILE**　　システムのファイルテーブルがいっぱい
- **EFAULT**　　ファイルデスクリプタが無効

pipeから返される2つのファイルデスクリプタは、特別な方法で接続されます。具体的には、

file_descriptor[1]に書き込まれたデータをfile_descriptor[0]から読み出すことができます。データは先入れ先出し方式で処理されます。先入れ先出しとは、First In, First Out（先に入ったものが先に出る）のことで、しばしばFIFOと略記されます。つまり、file_descriptor[1]にバイト1、2、3を順に書き込むと、file_descriptor[0]からの読み取りでも同じ順序で1、2、3が返されます。ちなみに、これとは異なる方式として後入れ先出し、つまりLIFO（Last In, First Out）という方式もあります。LIFOは主にスタックで使われます。

> **注意**
> pipeで返されるのは、ファイルストリームではなくファイルデスクリプタであることに注意してください。つまり、データにアクセスするには、freadやfwriteではなく、低水準のreadとwriteを使う必要があります。

さっそくpipeを使ってサンプルプログラムを作成してみましょう。プログラムの名前はpipe1.cとします。

例題 pipe関数

プログラムの中で、pipe関数に渡すパラメータとしてfile_pipesポインタを使っている点に注意してください。

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main()
{
    int data_processed;
    int file_pipes[2];
    const char some_data[] = "123";
    char buffer[BUFSIZ + 1];

    memset(buffer, '\0', sizeof(buffer));

    if (pipe(file_pipes) == 0) {
        data_processed = write(file_pipes[1], some_data, strlen(some_data));
        printf("Wrote %d bytes\n", data_processed);
        data_processed = read(file_pipes[0], buffer, BUFSIZ);
        printf("Read %d bytes: %s\n", data_processed, buffer);
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

プログラムを実行すると、次のように出力されます。

```
$ pipe1
Wrote 3 bytes
Read 3 bytes: 123
```

解説

2つのファイルデスクリプタ`file_pipes[]`を使って`pipe`を呼び出し、パイプを作成します。次に、ファイルデスクリプタ`file_pipes[1]`を使ってパイプにデータを書き込み、`file_pipes[0]`からデータを読み取ります。パイプにはいくらかの内部バッファがあり、`write`と`read`の呼び出しの間、データはこのバッファに格納されます。

`pipe`関数では、`file_descriptor[0]`への書き込みや`file_descriptor[1]`からの読み取りは未定義となっています。筆者のマシンでは-1が返されましたが、具体的な動作はシステムによって異なる可能性があります。

さて、上のサンプルプログラムでは、パイプを使う利点はたいしてありません。パイプが真価を発揮するのは2つのプロセス間でデータを受け渡すときです。前の章で見たように、プログラムの中から`fork`を使って新しいプロセスを作成した場合、オープンされていたファイルデスクリプタは、作成された新しいプロセスでもオープンされたままになります。もとのプロセスでパイプを作成し、`fork`で新しいプロセスを開始すると、パイプを介してこれら2つのプロセス間でデータをやりとりすることができます。

例題 パイプとfork

1 プログラムの名前は`pipe2.c`とします。最初の部分は`pipe1.c`と同じですが、今度は`fork`を呼び出します。`fork`からの戻り値を調べ、失敗した場合にはプログラムを終了します。

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main()
{
    int data_processed;
    int file_pipes[2];
    const char some_data[] = "123";
    char buffer[BUFSIZ + 1];
    int fork_result;

    memset(buffer, '\0', sizeof(buffer));

    if (pipe(file_pipes) == 0) {
        fork_result = fork();
        if (fork_result == -1) {
            fprintf(stderr, "Fork failure");
```

11

```
            exit(EXIT_FAILURE);
        }
```

2 forkからの戻り値fork_resultが0なら、子プロセスです。

```
        if (fork_result == 0) {
            data_processed = read(file_pipes[0], buffer, BUFSIZ);
            printf("Read %d bytes: %s\n", data_processed, buffer);
            exit(EXIT_SUCCESS);
        }
```

3 それ以外の場合は親プロセスです。

```
        else {
            data_processed = write(file_pipes[1], some_data,
                                   strlen(some_data));
            printf("Wrote %d bytes\n", data_processed);
        }
    }
    exit(EXIT_SUCCESS);
}
```

プログラムを実行すると、pipe1と同じ結果が出力されます。

```
$ pipe2
Wrote 3 bytes
Read 3 bytes: 123
```

解説

このプログラムでは、まずpipeを呼び出してパイプを作成し、次にforkを呼び出して新しいプロセスを作成しています。forkが成功した場合、親プロセスはパイプにデータを書き込み、子プロセスはパイプからデータを読み出します。親も子も、writeとreadをそれぞれ1回呼び出したあとに終了します。

pipeを使った最初のサンプルプログラムと結果は同じですが、2つの独立したプロセス間で読み書きが可能になった点で、かなり高度な機能を実現しています。このプログラムのしくみを図に表すと、次のようになります。

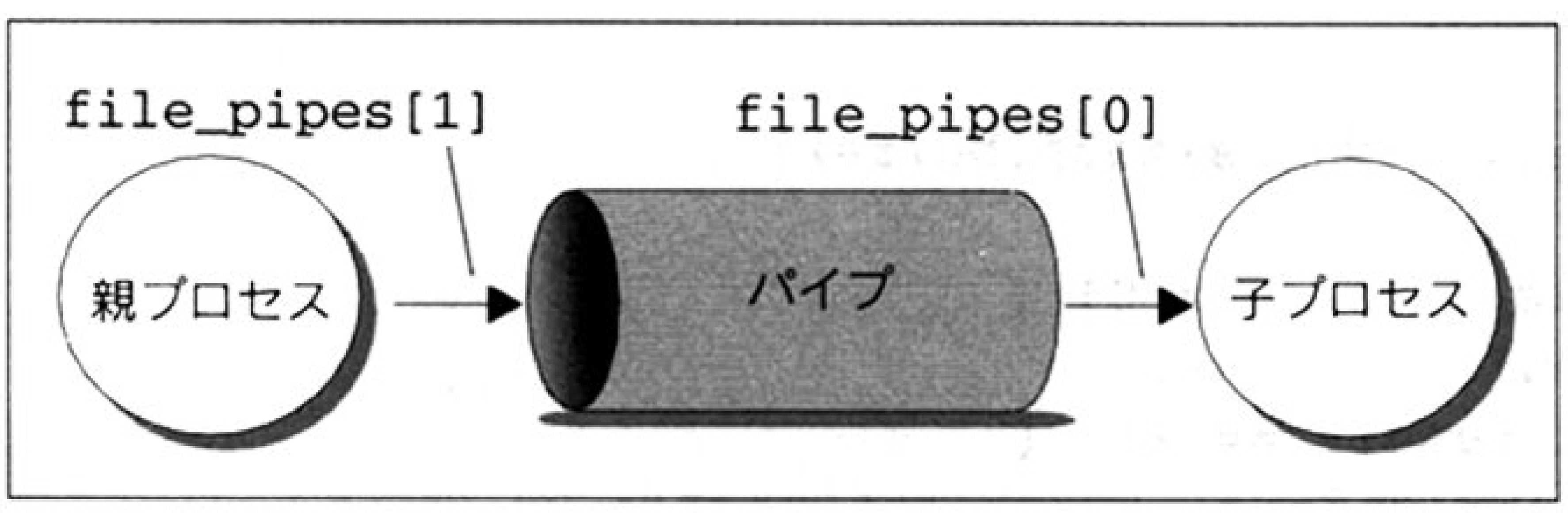


図11.2 親プロセスと子プロセスの間のパイプ

11.4 親プロセスと子プロセス

さて、サンプルプログラムで実現したことをさらに進めて、子プロセスが単に親と異なる"プロセス"の場合だけでなく、親と異なる"プログラム"の場合でもパイプを利用することはできないでしょうか。もちろん可能です。それには**exec**を使います。ただし、ひとつ問題があります。それは、**exec**で起動された新しいプロセスの側で、アクセスすべきファイルデスクリプタを知る必要があることです。前のサンプルプログラムでは、子プロセスは**file_pipes**のコピーにアクセスできるので、この点は問題になりませんでした。しかし、**exec**を使う場合、以前のプロセスは新しい子プロセスによって置き換えられてしまい、**file_pipes**のコピーにアクセスする方法は使用できません。この問題を回避するには、**exec**で起動するプログラムに目的のファイルデスクリプタを引数として渡します。

実際に動作を確かめるには、2つのプログラムが必要です。ひとつは**データ生産者**です。データ生産者は、パイプを作成し、**データ消費者**となる子プロセスを呼び出します。

例題 パイプとexec

1 **pipe2.c**を少し修正し、データ生産者となる最初のプログラムを作成して**pipe3.c**とします。

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main()
{
    int data_processed;
    int file_pipes[2];
    const char some_data[] = "123";
    char buffer[BUFSIZ + 1];
    int fork_result;

    memset(buffer, '\0', sizeof(buffer));

    if (pipe(file_pipes) == 0) {
        fork_result = fork();
        if (fork_result == -1) {
            fprintf(stderr, "Fork failure");
            exit(EXIT_FAILURE);
        }

        if (fork_result == 0) {
            sprintf(buffer, "%d", file_pipes[0]);
            (void)execl("pipe4", "pipe4", buffer, (char *)0);
            exit(EXIT_FAILURE);
```

```
        }
        else {
            data_processed = write(file_pipes[1], some_data,
                                   strlen(some_data));
            printf("%d - wrote %d bytes\n", getpid(), data_processed);
        }
    }
    exit(EXIT_SUCCESS);
}
```

2 次に、データ消費者となる簡単なプログラム**pipe4.c**を作成します。

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
    int data_processed;
    char buffer[BUFSIZ + 1];
    int file_descriptor;

    memset(buffer, '\0', sizeof(buffer));
    sscanf(argv[1], "%d", &file_descriptor);
    data_processed = read(file_descriptor, buffer, BUFSIZ);

    printf("%d - read %d bytes: %s\n", getpid(), data_processed, buffer);
    exit(EXIT_SUCCESS);
}
```

pipe3を実行すると、**pipe4**プログラムが呼び出され、次のように出力されます。

```
$ pipe3
980 - wrote 3 bytes
981 - read 3 bytes: 123
```

解説

最初の部分は**pipe2**プログラムと同じで、**pipe**を呼び出してパイプを作成し、**fork**を使って新しいプロセスを作成します。次に、**sprintf**を使ってパイプの読み取り側のファイルデスクリプタの値をバッファに保存し、これを**pipe4**への引数として渡します。

pipe4プログラムの呼び出しには**execl**を使っています。指定する引数は次のとおりです。

- 呼び出すプログラム
- **argv[0]**(プログラム名)

- **argv[1]**（呼び出すプログラムが読み取りのために使うファイルデスクリプタの値）
- **(char *)0**（引数の終わり）

pipe4プログラムは、渡された引数からファイルデスクリプタを取り出し、これを使ってデータを読み出します。

11.4.1 クローズされたパイプの読み取り

先へ進む前に、オープンされたファイルデスクリプタの扱いについて少し詳しく見ておきましょう。これまでのサンプルプログラムでは、単に読み手側のプロセスでデータを読み取ったあとに終了するだけでした。この方法では、プロセスの終了時にシステムによってファイルのクリーンアップ処理が行われることを期待しています。

しかし、標準入力からデータを読み取るほとんどのプログラムは、サンプルプログラムとは異なる方法で処理を行います。標準入力からデータを読み取るプログラムの場合、読み取るべきデータがどれくらいあるかはわかりません。したがって、この種のプログラムでは、一般にループの中でデータを読み取って処理し、読み取るべきデータがなくなるまでこの操作を繰り返します。

通常、**read**の呼び出しは**ブロック**されます。つまり、**read**を呼び出すプロセスは、データが利用可能になるまで待たされます。パイプのもう一端（書き込み側）がクローズされていた場合、パイプにはそれ以上データが書き込まれることはありません（いったんクローズされたパイプが再度オープンされることはありません）。こうしたパイプに対する読み取りが無期限に待たされないようにするために、書き込み用にオープンされていないパイプに対して**read**を呼び出すと、0が返されます。読み取り側のプロセスは、このしくみによって永久にブロックされることがないようになっています。補足すると、これは無効なファイルデスクリプタに対する読み取りとは異なります。ファイルデスクリプタが無効の場合、**read**はこれをエラーとみなして -1を返します。

forkをまたいでパイプを使う場合、パイプへの書き込みに使用できるファイルデスクリプタは、親プロセスのものと子プロセスのものとの2つがあります。パイプに書き込むためのこれらのファイルデスクリプタは、親プロセスと子プロセスの両方でクローズしなければなりません。さもなければ、パイプがクローズされたとはみなされず、**read**の呼び出しも失敗しません。この点については、あとで**O_NONBLOCK**フラグとFIFOについて説明する際にもう一度詳しく取り上げます。

11.4.2 標準入出力としてのパイプ

空のパイプに対する呼び出しを失敗させる方法について説明したので、今度はパイプを使って2つのプロセスを接続するもっとスマートな方法を示しましょう。それには、パイプのファイルデスクリプタとして、標準入力の0、標準出力の1などのようによく知られた値を使います。この場合、親プロセス側の処理はやや複雑になりますが、子プロセス側のプログラムは非常にシンプルに記述できます。

この方法の最大の利点は、引数にファイルデスクリプタを期待するような特別なプログラムではなく、ごくふつうのプログラムを呼び出すことができる点です。この機能を実現するには、第3章で取り上げた**dup**関数を使います。

```
#include <unistd.h>

int dup(int file_descriptor);
int dup2(int file_descriptor_one, int file_descriptor_two);
```

dupは**open**同様、新しいファイルデスクリプタをオープンします。ただし、**dup**によって作成されるファイルデスクリプタは、既存のファイルデスクリプタと同じファイル（またはパイプ）を参照します。**dup**は、使われていない最小の値を新しいファイルデスクリプタとして返します。一方、**dup2**は、パラメータ**file_descriptor_two**と同じか、またはそれより大きいファイルデスクリプタのうち最初に利用可能なものを返します。

メモ

dupや**dup2**の代わりに、汎用的な**fcntl**と**F_DUPFD**を組み合わせて使う方法もありますが、ファイルデスクリプタを複製する用途に特化した**dup**のほうが使いやすいでしょう。実際のプログラムでも、**fcntl**と**F_DUPFD**の組み合わせより**dup**のほうがよく使われています。

では、プロセス間でデータをやりとりするために、この**dup**をどのように使うのでしょうか。ポイントは2つあります。ひとつは標準入力ファイルデスクリプタが常に0であること、そしてもうひとつは、**dup**が返す新しいファイルデスクリプタが常に、利用可能な最小のファイルデスクリプタであることです。このことを利用して、まずファイルデスクリプタ0をクローズし、**dup**を呼び出します。すると、**dup**が返す新しいファイルデスクリプタは0になります。この新しいファイルデスクリプタは既存のファイルデスクリプタを複製したものです。したがって、標準入力を使って、**dup**に渡したファイルデスクリプタを持つファイルまたはパイプにアクセスできるようになります。つまり、同じファイルまたはパイプに関連付けられたファイルデスクリプタが2つ作成され、その1つは標準入力になるわけです。

表11.1 closeとdupによるファイルデスクリプタの操作

ファイルデスクリプタの値	最初の状態	closeの呼び出し後	dupの呼び出し後
0	標準入力		パイプのファイルデスクリプタ
1	標準出力	標準出力	標準出力
2	標準エラー	標準エラー	標準エラー
3	パイプのファイルデスクリプタ	パイプのファイルデスクリプタ	パイプのファイルデスクリプタ

前のサンプルプログラムに戻って上のテクニックを応用してみることにしましょう。今度は、子プロセス側のプログラムの標準入力ファイルデスクリプタをパイプの読み手側のファイルデスクリプタに置き換えます。また、使わないファイルデスクリプタをクローズし、子プロセス側のプログラムがパイプからのデータの終わりを正しく検出できるようにします。

例題 パイプとdup

1 **pipe3.c**を修正して**pipe5.c**を作成します。

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main()
{
    int data_processed;
    int file_pipes[2];
    const char some_data[] = "123";
    int fork_result;

    if (pipe(file_pipes) == 0) {
        fork_result = fork();
        if (fork_result == -1) {
            fprintf(stderr, "Fork failure");
            exit(EXIT_FAILURE);
        }

        if (fork_result == 0) {
            close(0);
            dup(file_pipes[0]);
            close(file_pipes[0]);
            close(file_pipes[1]);

            execlp("od", "od", "-c", (char *)0);
            exit(EXIT_FAILURE);
        }
        else {
            close(file_pipes[0]);
            data_processed = write(file_pipes[1], some_data,
                                   strlen(some_data));
```

```
                close(file_pipes[1]);
                printf("%d - wrote %d bytes\n", getpid(), data_processed);
            }
        }
        exit(EXIT_SUCCESS);
    }
```

これはサンプルプログラムなので、本来ならもっと厳密にすべきエラーチェックを省いています。プログラムを実行すると、次のように出力されます。

```
$ pipe5
1239 - wrote 3 bytes
0000000   1   2   3
0000003
```

解説

以前と同様、このプログラムでもパイプを作成し、**fork**を呼び出して子プロセスを作成しています。この時点で、親プロセスと子プロセスは、パイプにアクセスするためのファイルデスクリプタを読み取り用と書き込み用にそれぞれ1つずつ持っています。したがって、これらを合計すると、全部で4つのファイルデスクリプタがオープンされていることになります。

最初に子プロセスに注目しましょう。子プロセスでは、**close(0)**を実行して標準入力をクローズし、次に**dup(file_pipes[0])**を呼び出します。これで、パイプの読み手側に関連付けられているファイルデスクリプタがファイルデスクリプタ0、つまり標準入力として複製されます。このあと、子プロセスは、パイプからデータを読み取るためのもとのファイルデスクリプタ**file_pipes[0]**をクローズします。また、子プロセスではパイプへの書き込みは行わないので、パイプに関連付けられている書き手側のファイルデスクリプタ**file_pipes[1]**もクローズします。これで、子プロセスが持つファイルデスクリプタは、パイプに関連付けられたファイルデスクリプタ、つまりファイルデスクリプタ0（標準入力）だけになります。

子プロセスは次に**exec**を使って、標準入力を読み取るプログラムを呼び出します。サンプルプログラムでは**od**コマンドを使っています。**od**コマンドは、ちょうど端末からのユーザー入力を待つように、データが利用可能になるのを待ちます。実際には、両者の違いを見いだすための特別なコードを意図的に記述しないかぎり、入力がパイプと端末のどちらからきたものなのかはわかりません。

親プロセスの側ではパイプを読み取ることはないので、読み手側の**file_pipes[0]**をクローズします。次に、パイプにデータを書き込みます。データをすべて書き込んだら、親プロセスは終了します。これで、パイプに書き込むことのできるファイルデスクリプタは存在しなくなり、**od**プログラムはパイプに書き込まれた3バイトを読み取ることができます。**od**コマンドがさらにパイプから読み込もうとすると0が返されるので、ファイルの終わりに達したことがわかります。**od**プログラムは、**read**が0を返すと終了します。これは、端末で**od**コマンドを実行し、ファイ

ルの終わりを知らせるのに**Ctrl - D**を入力したのと結果的に同じです。

以上の手順を図にしてみましょう。まず、**pipe**を呼び出したあとは、次のように自分自身にパイプが接続されています。

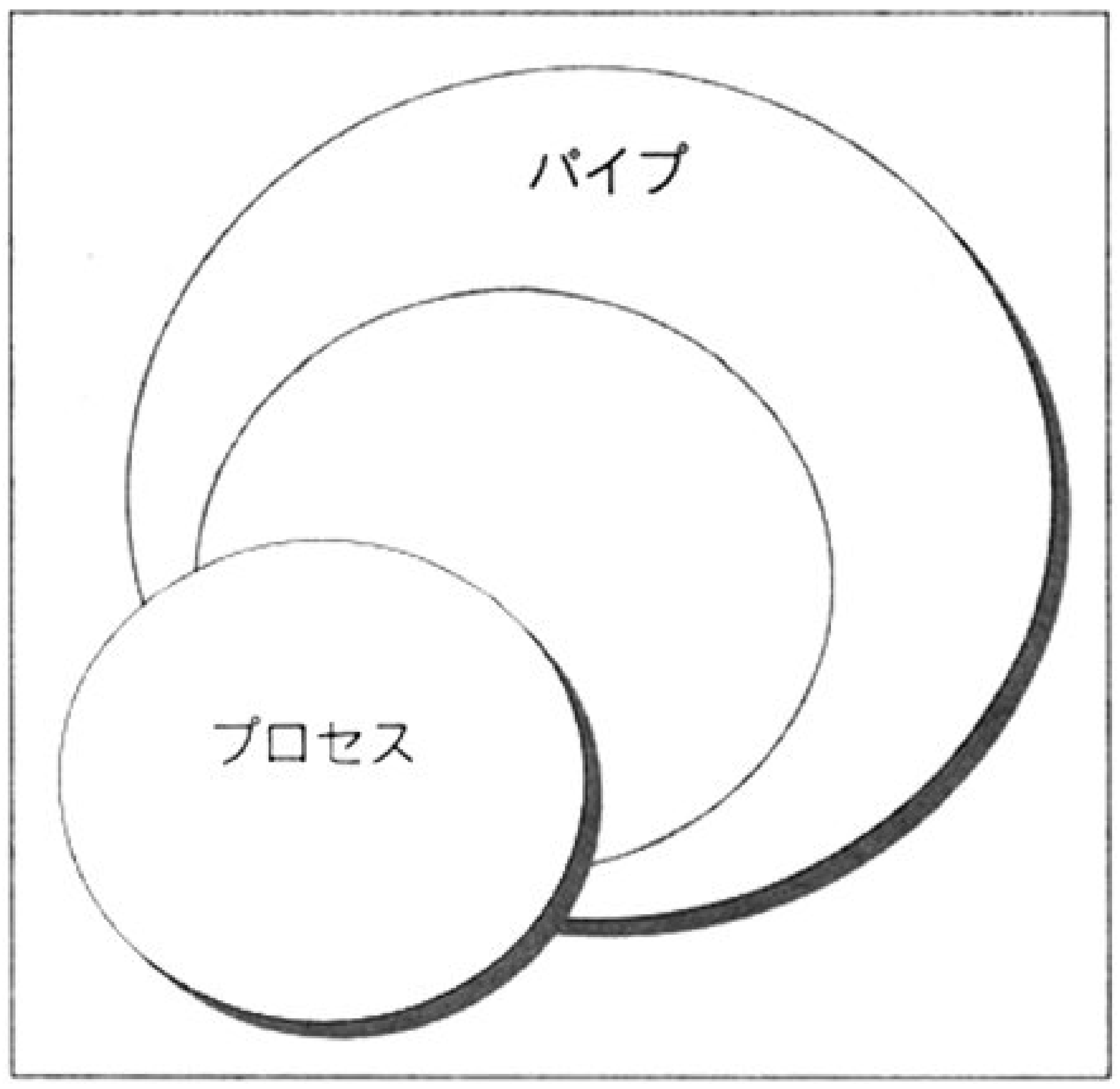


図11.3 pipeの呼び出し後の状態

次は、**fork**を呼び出したあとの状態です。2つのプロセスは1つのパイプで接続されていますが、パイプの両端には読み取り用と書き込み用の2つの"口"が開いたままです。

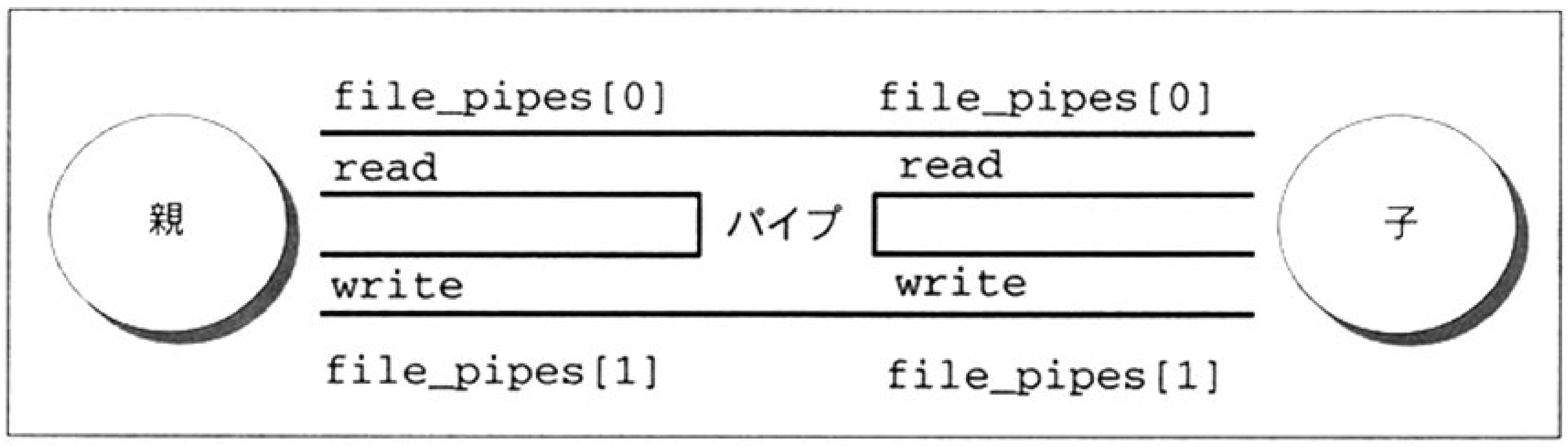


図11.4 forkの呼び出し後の状態

次は、パイプを介したデータ転送の準備ができた状態です。パイプの両端でそれぞれ口が1つ閉じられ、左から右へと一方向の流れができます。

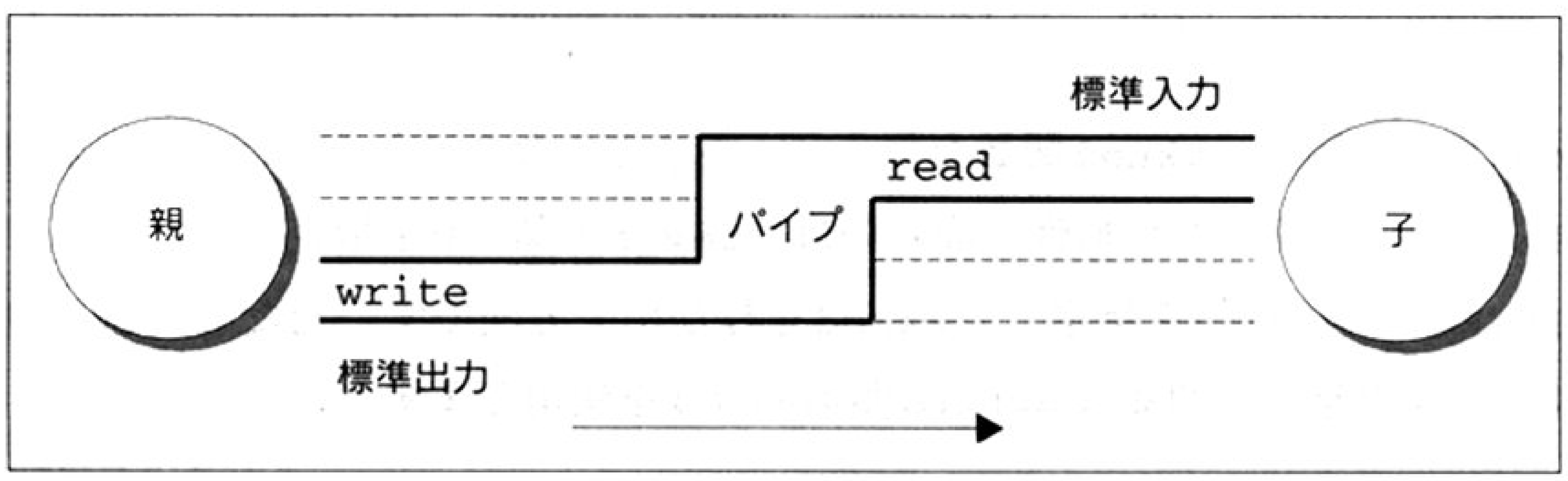


図11.5 データ転送の準備ができた状態

11.5 名前付きパイプ（FIFO）

これまでに取り上げたのは、互いに関連のあるプログラム、つまり共通の祖先を持つプログラムの間でデータをやりとりするケースだけでした。しかし、現実には、互いに関連のないプログラムの間でのデータの交換が必要になる場合もあります。

このような場合に利用できるのが**FIFO**です。FIFOは、しばしば**名前付きパイプ**とも呼ばれます。名前付きパイプは、ファイルシステム内に1つの名前として存在する特別な種類のファイルですが（UNIXではすべてのものがファイルとして扱われます）、これまでに取り上げてきた名前のないパイプと同じようにふるまいます。

名前付きパイプは、コマンドラインやプログラムの中から作成することができます。コマンドラインでは、従来、次のように**mknod**コマンドを使って作成するのが一般的でした。

```
$ mknod filename p
```

ただし、この**mknod**コマンドはX/Openのコマンドリストに含まれていません。このため、UNIXシステムによっては利用できないことがあります。現在推奨されているのは、次のようなコマンドを使う方法です。

```
$ mkfifo filename
```

注意

一部の古いバージョンのUNIXでは、**mknod**コマンドしか利用できないことがあります。Linuxでは、**mknod**と**mkfifo**のどちらのコマンドもサポートされています。

プログラムの中から名前付きパイプを作成する場合には、次の2つの関数を使用できます。

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(const char *filename, mode_t mode);
int mknod(const char *filename, mode_t mode | S_IFIFO, (dev_t) 0);
```

mknodコマンド同様、**mknod**関数を使うと、さまざまなスペシャルファイルを作成することができます。上の2つのうち移植性が高いのは、**dev_t**の値に0を指定し、ファイルアクセスモードと**S_IFIFO**との論理和を指定して**mknod**を呼び出す方法です。ただし、以下のサンプルプログラムでは、より構文の単純な**mkfifo**関数のほうを使用します。

例題 名前付きパイプの作成

fifo1.cという名前で次のプログラムを作成します。

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>

int main()
{
    int res = mkfifo("/tmp/my_fifo", 0777);
    if (res == 0) printf("FIFO created\n");
    exit(EXIT_SUCCESS);
}
```

プログラムを実行すると、次のようにパイプが作成されます。

```
$ ls -lF /tmp/my_fifo
prwxr-xr-x   1 rick     users          0 Dec 10 14:55 /tmp/my_fifo|
```

lsからの出力で、最初の文字がパイプを表すpになっていることに注目してください。また、最後の文字|は、lsコマンドの-Fオプションによって付加された文字で、これもファイルがパイプであることを示しています。

解説

このプログラムでは、mkfifo関数を使ってスペシャルファイルを作成しています。作成時のモードには0777を指定していますが、作成されたファイルのモードは755になっています。これは、ユーザーマスク(umask)の設定(上の場合には022)によってモードが変更されたためです。

FIFOを削除するには、rmコマンドを使います。プログラムの中から削除する場合には、unlink関数を使います。

11.5.1 FIFOへのアクセス

名前付きパイプの優れた点のひとつは、ファイルシステム内に存在するので、通常のファイルを扱うのと同様にコマンドラインで名前付きパイプを使用できることです。FIFOファイルを使ったプログラミングに進む前に、通常のコマンドを使ってFIFOファイルの動作を見ておきましょう。

例題 FIFOファイルへのアクセス

1 まず、(空の) FIFOから読み出してみます。

```
$ cat < /tmp/my_fifo
```

2 次に、FIFOに書き込んでみます。

```
$ echo "sdsdfasdf" > /tmp/my_fifo
```

3 上の操作を両方同時に行うと、パイプを介した情報の受け渡しが可能になります。

```
$ cat < /tmp/my_fifo &
[1] 1316
$ echo "sdsdfasdf" > /tmp/my_fifo
sdsdfasdf

[1]+  Done                    cat </tmp/my_fifo
$
```

最初の2つの手順では、割り込みをかけるまで(**Ctrl - C**を入力するまで)、プログラムはハングします。最後の手順では、画面に出力が表示されます。

解説

最初の2つの手順ではFIFOにデータが入っていないので、catの場合はデータが利用できるようになるまで、またechoの場合はほかのプロセスがデータを読み出すまでブロックされます。

3つ目の手順では、catプロセスがまずバックグラウンドでブロックされた状態になります。echoによってデータが利用できる状態になると、catコマンドはデータを読み取って標準出力に表示します。このとき、catコマンドは次のデータを待たずにすぐに終了します。catコマンドがブロックされないのは、パイプがクローズされ、catプログラムから呼び出されたreadが、ファイルの終わりに達したことを示す0を返すためです。

コマンドラインプログラムを使ってアクセスする場合のFIFOのふるまいについては、以上のとおりです。今度は、FIFOのプログラミングインタフェースについてさらに詳しく見ていきましょう。プログラミングインタフェースでは、FIFOにアクセスする場合のreadとwriteの動作を細かく指定することができます。

> **注意**
>
> pipe関数で作成するパイプとは異なり、FIFOは(オープンされたファイルデスクリプタではなく)名前を持ったファイルとして存在しています。このため、FIFOに対して読み書きを行うには、まずFIFOをオープンしなければなりません。FIFOをオープンまたはクローズするには、通常のファイルの場合と同様にopen関数とclose関数を使用でき、いくつかの追加の機能も利用できます。openを呼び出すときには、通常のファイルの代わりにFIFOのパス名を指定します。

11.5.2 openを使ったFIFOのオープン

FIFOをオープンする場合、モードにO_RDWRを指定して読み書き用にオープンすることができないという制限があります（読み書き用にオープンした場合の結果は未定義です）。ただし、この制限は実際には妥当な制限といえます。FIFOを使ってデータを渡す場合、通常はデータの流れが一方向になり、O_RDWRモードを使う必要はないからです。もし読み書き用にオープンできるなら、プロセスは自分自身の出力をパイプから読み取ることになります。

2つのプログラムの間で両方向でデータをやりとりしたい場合には、各方向ごとにFIFOまたはパイプを用意するか、あるいは1つのFIFOをクローズして再度オープンすることによって明示的にデータフローの向きを変えるのがよい方法です。FIFOを使った両方向でのデータのやりとりについては、あとでもう一度取り上げます。

FIFOと通常のファイルには、もうひとつ違いがあります。それは、open関数の第2パラメータopen_flagで指定するO_NONBLOCKオプションに関するものです。openのモードでこのオプションを指定した場合、openの呼び出しがどのように処理されるかだけでなく、openから返されたファイルデスクリプタに対するread要求とwrite要求がどのように処理されるかも影響を受けます。

openを呼び出すときに指定するO_RDONLY、O_WRONLY、O_NONBLOCKの各フラグの有効な組み合わせは、全部で4とおりあります。ひとつひとつ具体的に見ていきましょう。

```
open(const char *path, O_RDONLY);
```

この場合、openの呼び出しはブロックされます。つまり、なんらかのプロセスが同じFIFOを書き込み用にオープンするまで戻りません。これは、コマンドラインからFIFOを使う例で示したcatのケースと同じです。

```
open(const char *path, O_RDONLY | O_NONBLOCK);
```

この場合、FIFOがほかのプロセスによって書き込み用にオープンされていなくても、openは成功し、ただちに戻ります。

```
open(const char *path, O_WRONLY);
```

この場合、なんらかのプロセスが同じFIFOを読み取り用にオープンするまで、openの呼び出しはブロックされます。

```
open(const char *path, O_WRONLY | O_NONBLOCK);
```

この場合、openの呼び出しはただちに戻ります。ただし、読み取り用にFIFOをオープンして

いるプロセスがない場合、openはエラー（-1）を返し、FIFOはオープンされません。読み取り用にFIFOをオープンしているプロセスがある場合には、openから返されたファイルデスクリプタをFIFOへの書き込みに使用することができます。

O_RDONLYとO_WRONLYでは、O_NONBLOCKの使い方が異なることに注意してください。ブロックされない書き込み用のopenは、読み取り用にパイプをオープンしているプロセスがなければ失敗しますが、ブロックされない読み取り用のopenのほうは失敗しません。なお、close関数の動作については、O_NONBLOCKフラグは影響しません。

O_NONBLOCKフラグを指定した場合のopenの動作を利用して2つのプロセスを同期させてみましょう。ここでは複数のプログラムを使わずに、1つのテストプログラムでFIFOの動作を確かめます。作成するプログラムの名前はfifo2.cとします。

例題 FIFOファイルのオープン

1 必要なヘッダーファイルをインクルードし、FIFOファイルの名前を定義したあと、コマンドライン引数の数をチェックします。

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

#define FIFO_NAME "/tmp/my_fifo"

int main(int argc, char *argv[])
{
    int res;
    int open_mode = 0;

    if (argc < 2) {
        fprintf(stderr, "Usage: %s <some combination of\
               O_RDONLY O_WRONLY O_NONBLOCK\n", *argv);
        exit(EXIT_FAILURE);
    }
```

2 受け取った引数に基づいてopen_modeの値を設定します。

```
    argv++;
    if (strncmp(*argv, "O_RDONLY", 8) == 0) open_mode |= O_RDONLY;
    if (strncmp(*argv, "O_WRONLY", 8) == 0) open_mode |= O_WRONLY;
    if (strncmp(*argv, "O_NONBLOCK", 10) == 0) open_mode |= O_NONBLOCK;
    argv++;
    if (*argv) {
        if (strncmp(*argv, "O_RDONLY", 8) == 0) open_mode |= O_RDONLY;
```

```
        if (strncmp(*argv, "O_WRONLY", 8) == 0) open_mode |= O_WRONLY;
        if (strncmp(*argv, "O_NONBLOCK", 10) == 0) open_mode |= O_NONBLOCK;
    }
```

3 目的のFIFOが存在するかどうかを調べ、存在しない場合には作成します。次に、FIFOをオープンし、openの動作を確認できるようにメッセージを表示します。短時間スリープしたあと、最後にFIFOをクローズします。

```
    if (access(FIFO_NAME, F_OK) == -1) {
        res = mkfifo(FIFO_NAME, 0777);
        if (res != 0) {
            fprintf(stderr, "Could not create fifo %s\n", FIFO_NAME);
            exit(EXIT_FAILURE);
        }
    }

    printf("Process %d opening FIFO\n", getpid());
    res = open(FIFO_NAME, open_mode);
    printf("Process %d result %d\n", getpid(), res);
    sleep(5);
    if (res != -1) (void)close(res);
    printf("Process %d finished\n", getpid());
    exit(EXIT_SUCCESS);
}
```

解説

このプログラムでは、O_RDONLY、O_WRONLY、O_NONBLOCKの組み合わせをコマンドラインで指定することができます。具体的には、フラグを表す文字列とコマンドラインパラメータとを比較し、一致した文字列に対応するフラグを設定します。access関数を使ってFIFOファイルがすでに存在するかどうかを調べ、存在しない場合には作成します。

FIFOファイルは削除しません。これは、ほかのプログラムがFIFOを使っているかどうか調べる方法がないためです。では、実際にさまざまな組み合わせを試してみましょう。

◆ O_RDONLYと、O_NONBLOCKを指定しないO_WRONLYとの組み合わせ

```
$ fifo2 O_RDONLY &
[1] 152
Process 152 opening FIFO
$ fifo2 O_WRONLY
Process 153 opening FIFO
Process 152 result 3
Process 153 result 3
Process 152 finished
Process 153 finished
```

これは、名前付きパイプの最も一般的な使い方です。まず、読み手側のプロセスが起動してopenの呼び出しで待機します。もうひとつのプログラムがFIFOをオープンすると、2つのプログラムが両方とも処理を続けます。読み手側と書き手側の両方のプロセスが、openを呼び出した時点で同期していることに注目してください。

> **メモ** UNIXでは、ブロックされているプロセスはCPUリソースを消費しません。したがって、上のようなプロセス同期の方法はCPU効率の点でも優れています。

◆ O_NONBLOCKを指定したO_RDONLYと、O_WRONLYとの組み合わせ

```
$ fifo2 O_RDONLY O_NONBLOCK &
[1] 160
Process 160 opening FIFO
Process 160 result 3
$ fifo2 O_WRONLY
Process 161 opening FIFO
Process 161 result 3
Process 160 finished
Process 161 finished
[1]+  Done                    fifo2 O_RDONLY O_NONBLOCK
```

今度は、読み手側のプロセスはopenを実行し、書き手側のプロセスが存在しない場合でもそのまま処理を続けます。書き手側でも、FIFOがすでに読み取り用にオープンされているので、openの呼び出し後、処理がそのまま続けられます。

以上の2つの組み合わせは、FIFOを使う場合のopenのモードとして最も一般的なケースです。そのほかの組み合わせについても試してみてください。

11.5.3 FIFOに対する読み取りと書き込み

O_NONBLOCKモードは、FIFOに対するreadとwriteの動作に影響を与えます。

ブロックされるFIFO（O_NONBLOCKの指定なしにオープンされたFIFO）のデータが空の場合、このFIFOに対してreadを呼び出すと、データが利用可能になるまで待たされます。一方、ブロックされないFIFOのデータが空の場合、このFIFOに対してreadを呼び出すと、0バイトが返されます。

ブロックされるFIFOのデータがいっぱいの場合、このFIFOに対してwriteを呼び出すと、データが書き込めるようになるまで待たされます。一方、書き込まれるバイトのうち一部しか受け付けることのできないFIFOに対してwriteを呼び出した場合、結果は次のいずれかになります。

- PIPE_BUFバイトまたはそれより少ないバイト数の書き込み要求の場合、データを書き込むことができなければ、失敗します。
- PIPE_BUFバイトより大きい書き込み要求の場合、データの一部を書き込みます。この場合には、実際に書き込まれたバイト数を返します。返されるバイト数は0になることもあります。

FIFOのサイズは重要です。ある時点でFIFOに存在することができるデータの量は、システムによって制限されています。通常、この制限は、**limits.h**内で**#define PIPE_BUF**によって定義されています。この値は、Linuxやその他の多くのUNIXシステムでは4096バイトですが、システムによっては512バイトしかない場合もあります。**O_WRONLY**でオープンされたFIFO、つまりブロックされるFIFOに対する**PIPE_BUF**バイト以下の書き込みは、すべてのバイトが書き込まれるか、またはまったく書き込まれないかのいずれかになることがシステムによって保証されています。

FIFOのサイズに関する制限は、FIFOの読み手と書き手がそれぞれ1つだけのような単純なケースではそれほど重要ではありません。しかし、現実のプログラミングでは、1つのFIFOを使用し、異なる複数のプログラムから単独の読み手に対して要求を送信できるようにするほうが一般的です。こうしたケースでは、いくつかの異なるプログラムが同時にFIFOに書き込もうとした場合に、それぞれのプログラムからのデータが混じり合わないようにすること、つまりそれぞれの**write**操作をアトミックにする必要があります。

FIFOに対するアトミックな書き込みを実現するには、すべての**write**要求が、ブロックされるFIFOを対象として行われるようにして、書き込むデータのサイズを**PIPE_BUF**バイトより少なくします。こうすると、データが混じり合わないことがシステムによって保証されます。単独の書き手と単独の読み手しかいない場合は別ですが、一般にFIFOを介したデータのやりとりは、**PIPE_BUF**バイトのブロックごとに行うのがよい方法です。

以下では、互いに関連のないプロセスの間で名前付きパイプを使って通信する方法を示すために、**fifo3.c**と**fifo4.c**の2つのプログラムを作成します。

例題 FIFOを使ったプロセス間通信

1 最初に示すのは、**fifo2.c**に修正を加えた生産者プログラム**fifo3.c**です。このプログラムは、パイプが存在しなければ作成し、可能なかぎりすみやかにデータを書き込みます。なお、やりとりするデータの内容はこのプログラムの目的とは関係ないので、**buffer**の初期化は行っていません。

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
```

11

```
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>

#define FIFO_NAME "/tmp/my_fifo"
#define BUFFER_SIZE PIPE_BUF
#define TEN_MEG (1024 * 1024 * 10)

int main()
{
    int pipe_fd;
    int res;
    int open_mode = O_WRONLY;
    int bytes_sent = 0;
    char buffer[BUFFER_SIZE + 1];

    if (access(FIFO_NAME, F_OK) == -1) {
        res = mkfifo(FIFO_NAME, 0777);
        if (res != 0) {
            fprintf(stderr, "Could not create fifo %s\n", FIFO_NAME);
            exit(EXIT_FAILURE);
        }
    }

    printf("Process %d opening FIFO O_WRONLY\n", getpid());
    pipe_fd = open(FIFO_NAME, open_mode);
    printf("Process %d result %d\n", getpid(), pipe_fd);

    if (pipe_fd != -1) {
        while(bytes_sent < TEN_MEG) {
            res = write(pipe_fd, buffer, BUFFER_SIZE);
            if (res == -1) {
                fprintf(stderr, "Write error on pipe\n");
                exit(EXIT_FAILURE);
            }
            bytes_sent += res;
        }
        (void)close(pipe_fd);
    }
    else {
        exit(EXIT_FAILURE);
    }

    printf("Process %d finished\n", getpid());
    exit(EXIT_SUCCESS);
}
```

2 次に示すのは、消費者プログラム`fifo4.c`です。FIFOからデータを読み出します（実際には読み捨てるだけです）。

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>

#define FIFO_NAME "/tmp/my_fifo"
#define BUFFER_SIZE PIPE_BUF

int main()
{
    int pipe_fd;
    int res;
    int open_mode = O_RDONLY;
    char buffer[BUFFER_SIZE + 1];
    int bytes_read = 0;

    memset(buffer, '\0', sizeof(buffer));

    printf("Process %d opening FIFO O_RDONLY\n", getpid());
    pipe_fd = open(FIFO_NAME, open_mode);
    printf("Process %d result %d\n", getpid(), pipe_fd);

    if (pipe_fd != -1) {
        do {
            res = read(pipe_fd, buffer, BUFFER_SIZE);
            bytes_read += res;
        } while (res > 0);
        (void)close(pipe_fd);
    }
    else {
        exit(EXIT_FAILURE);
    }

    printf("Process %d finished, %d bytes read\n", getpid(), bytes_read);
    exit(EXIT_SUCCESS);
}
```

2つのプログラムを同時に実行します。読み手側のプログラムについては、実行時間を測定するために**time**コマンドを使います。出力例を次に示します（一部省略しています）。

```
$ fifo3 &
[1] 375
Process 375 opening FIFO O_WRONLY
$ time fifo4
Process 377 opening FIFO O_RDONLY
Process 375 result 3
Process 377 result 3
Process 375 finished
```

```
Process 377 finished, 10485760 bytes read
0.00user 0.13system 0:00.19elapsed 65%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (56major+8minor)pagefaults 0swaps
[1]+  Done                    fifo3
```

解説

fifo3と**fifo4**のどちらのプログラムも、ブロックされるモードでFIFOを使っています。最初に実行する書き手／生産者側の**fifo3**は、読み手がFIFOをオープンするまでブロックされます。消費者側の**fifo4**が起動すると、ブロックは解除され、書き手側はパイプへのデータの書き込みを開始します。同時に、読み手側がパイプからのデータの読み取りを開始します。

> **メモ**
> 2つのプロセスはいずれも、実行可能なときに実行され、実行できないときにはブロックされるようにスケジュールされます。したがって、パイプがいっぱいの場合には書き手側がブロックされ、パイプが空の場合には読み手側がブロックされます。

timeコマンドの出力を見ると、読み手側のプログラムの実行と10MBのデータの読み取りに0.19秒しかかかっていません。このことから、少なくとも筆者が使っているバージョンのLinuxでは、パイプがプログラム間でデータを転送するための効率的な方法であることがわかります。パイプを使って大量のデータをやりとりする場合には、プログラムで実装したパイプの効率をテストし、目的に見合ったパフォーマンスが出ているかどうかチェックするとよいでしょう。

11.6 FIFOを使ったクライアントサーバーアプリケーション

最後に、名前付きパイプを使ってごく単純なクライアントサーバーアプリケーションを構築する方法を示しましょう。サーバープロセスは、クライアントからの要求を受け取り、これを処理して結果を返します。

作成するプログラムでは、1つのサーバープロセスに対して複数のクライアントから要求を送信できるようにします。プログラムを簡潔にするために、処理するデータは**PIPE_BUF**バイトより小さいサイズのブロックに分割できるものとします。クライアントサーバーアプリケーションはさまざまな方法で実装できますが、ここでは名前付きパイプの使い方を説明するために、そのうちのひとつの方法を示します。

サーバーが同時に処理できる情報のブロックは1つだけなので、サーバーが読み取るFIFOは1つだけにして、このFIFOに対して各クライアントが書き込みを行うことにします。FIFOはブロックされるモードでオープンし、サーバーとクライアントが必要に応じて自動的にブロックされるようにします。

一方、処理済みのデータをクライアントに返す部分は少し複雑になり、データを返すためのパイプをクライアントごとに用意する必要があります。このため、もとのデータをサーバーに送るときにクライアントのプロセスIDを渡し、サーバーとクライアントの両方がパイプに対して一意の名前を生成できるようにします。

例題 名前付きパイプを使ったクライアントサーバーアプリケーション

1 クライアントとサーバーの両方に共通するデータを定義したヘッダーファイル`cliserv.h`を用意します。必要なシステムヘッダーファイルもここでインクルードします。

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>

#define SERVER_FIFO_NAME "/tmp/serv_fifo"
#define CLIENT_FIFO_NAME "/tmp/cli_%d_fifo"

#define BUFFER_SIZE 20

struct data_to_pass_st {
    pid_t  client_pid;
    char   some_data[BUFFER_SIZE - 1];
};
```

2 次は`server.c`プログラムです。まず、サーバー側のパイプを作成してオープンします。パイプは読み取り専用に設定し、ブロックされるモードでオープンします。動作を確認しやすいようにしばらくスリープしたあと、サーバーはクライアントからのデータ（`data_to_pass_st`構造体）を読み取ります。

```
#include "cliserv.h"
#include <ctype.h>

int main()
{
    int server_fifo_fd, client_fifo_fd;
    struct data_to_pass_st my_data;
    int read_res;
    char client_fifo[256];
    char *tmp_char_ptr;

    mkfifo(SERVER_FIFO_NAME, 0777);
    server_fifo_fd = open(SERVER_FIFO_NAME, O_RDONLY);
```

11

```
    if (server_fifo_fd == -1) {
        fprintf(stderr, "Server fifo failure\n");
        exit(EXIT_FAILURE);
    }

    sleep(10); /* lets clients queue for demo purposes */

    do {
        read_res = read(server_fifo_fd, &my_data, sizeof(my_data));
        if (read_res > 0) {
```

3 次に、クライアントから読み取ったデータを加工します。ここでは、メンバ`some_data`に含まれるすべての文字を大文字に変換します。また、受け取った`client_pid`と`CLIENT_FIFO_NAME`から、クライアントにデータを返すパイプの名前を作成します。

```
            tmp_char_ptr = my_data.some_data;
            while (*tmp_char_ptr) {
                *tmp_char_ptr = toupper(*tmp_char_ptr);
                tmp_char_ptr++;
            }
            sprintf(client_fifo, CLIENT_FIFO_NAME, my_data.client_pid);
```

4 パイプを書き込み専用、かつブロックされるモードでオープンし、処理したデータを送ります。最後に、ファイルをクローズし、FIFOを`unlink`してサーバーFIFOをシャットダウンします。

```
            client_fifo_fd = open(client_fifo, O_WRONLY);
            if (client_fifo_fd != -1) {
                write(client_fifo_fd, &my_data, sizeof(my_data));
                close(client_fifo_fd);
            }
        }
    } while (read_res > 0);
    close(server_fifo_fd);
    unlink(SERVER_FIFO_NAME);
    exit(EXIT_SUCCESS);
}
```

5 次は`client.c`です。最初の部分では、サーバーFIFOが存在していればファイルとしてオープンします。次に、サーバーに送信するデータの一部として使う自身のプロセスIDを取得し、クライアントFIFOを作成します。

```
#include "cliserv.h"
#include <ctype.h>

int main()
{
    int server_fifo_fd, client_fifo_fd;
    struct data_to_pass_st my_data;
```

```
    int times_to_send;
    char client_fifo[256];

    server_fifo_fd = open(SERVER_FIFO_NAME, O_WRONLY);
    if (server_fifo_fd == -1) {
        fprintf(stderr, "Sorry, no server\n");
        exit(EXIT_FAILURE);
    }

    my_data.client_pid = getpid();
    sprintf(client_fifo, CLIENT_FIFO_NAME, my_data.client_pid);
    if (mkfifo(client_fifo, 0777) == -1) {
        fprintf(stderr, "Sorry, can't make %s\n", client_fifo);
        exit(EXIT_FAILURE);
    }
```

6 ループを5回まわりながらクライアントデータをサーバーに送信します。サーバーにデータを送信したら、クライアントFIFOを読み取り専用、かつブロックされるモードでオープンし、書き込まれたデータを読み取ります。最後に、サーバーFIFOをクローズし、クライアントFIFOを削除します。

```
    for (times_to_send = 0; times_to_send < 5; times_to_send++) {
        sprintf(my_data.some_data, "Hello from %d", my_data.client_pid);
        printf("%d sent %s, ", my_data.client_pid, my_data.some_data);
        write(server_fifo_fd, &my_data, sizeof(my_data));
        client_fifo_fd = open(client_fifo, O_RDONLY);
        if (client_fifo_fd != -1) {
            if (read(client_fifo_fd, &my_data, sizeof(my_data)) > 0) {
                printf("received: %s\n", my_data.some_data);
            }
            close(client_fifo_fd);
        }
    }
    close(server_fifo_fd);
    unlink(client_fifo);
    exit(EXIT_SUCCESS);
}
```

このクライアントサーバーアプリケーションをテストするには、サーバーを1つ、クライアントを複数起動する必要があります。これらすべてをなるべく間をあけずに実行するために、次のシェルコマンドを使います。

```
$ server &
$ for i in 1 2 3 4 5
> do
> client &
> done
```

これで、サーバープロセスが1つ、クライアントプロセスが5つ起動します。クライアントからの出力は次のようになります（一部省略しています）。

```
531 sent Hello from 531, received: HELLO FROM 531
532 sent Hello from 532, received: HELLO FROM 532
529 sent Hello from 529, received: HELLO FROM 529
530 sent Hello from 530, received: HELLO FROM 530
531 sent Hello from 531, received: HELLO FROM 531
532 sent Hello from 532, received: HELLO FROM 532
```

クライアントからの要求は前後していますが、それぞれのクライアントには処理されたデータが適切に返されていることがわかります。なお、画面に表示されるメッセージの順序はマシンによって異なります。また、同じマシンでも、テストを実行するたびに異なる可能性があります。

解説

ここでは主にサーバーとクライアントが情報をやりとりする際の処理手順について説明します。

サーバー側では、読み取り専用、かつブロックされるモードでFIFOを作成します。このため、最初のクライアントがFIFOを書き込み用にオープンするまで、サーバーはブロックされます。最初のクライアントが接続すると、サーバープロセスのブロックが解除され、**sleep**が実行されます。これでクライアントからの書き込みが蓄積されます。（**sleep**を実行しているのは、同時に複数のクライアントが接続してもプログラムが正しく動作することを示すためです。実際のアプリケーションではこのような方法は使いません。）

クライアント側では、サーバーFIFOをオープンしたあと、サーバーから返されるデータを読み取るために一意の名前の付いたFIFOを作成します。次に、サーバーにデータを書き込みます。パイプがいっぱいだったり、サーバーがまだスリープ中だった場合、クライアントはブロックされます。その後、自分が作成したFIFOに対して**read**を呼び出し、サーバーからの応答を待ちます。

サーバープロセスは、クライアントからデータを受け取るとすぐに処理し、クライアントのパイプを書き込み用にオープンして（これでクライアントのブロックは解除されます）、データを書き込みます。クライアント側では、ブロックが解除された時点で、サーバーが書き込んだデータをパイプから読み出すことができます。

以上の動作は、最後のクライアントがサーバーのパイプをクローズするまで繰り返し実行されます。サーバーのパイプがクローズされると、パイプを書き込み用にオープンしているプロセスがなくなり、サーバー側の**read**は失敗します（0を返します）。この部分は、実際に実用になるプログラムを作成する場合、つまり最後のクライアントがサーバーのパイプをクローズしたあともほかのクライアントからの接続を待つようにする場合には、次のいずれかの方法で修正する必要があります。

- サーバーパイプに対する書き込み用のファイルデスクリプタをオープンし、readが（0を返すのではなく）常にブロックされるようにします。
- readが0バイトを返したら、サーバーパイプをクローズして再度オープンし、最初にサーバープロセスが起動したときと同様、クライアントがパイプをopenするまでサーバープロセスがブロックされるようにします。

クライアントからの接続を待ち続ける方法については、以下で名前付きパイプを使ってCDデータベースアプリケーションを書き直す際に具体的な実装例を示します。

11.7 CDデータベースアプリケーション

直前のサンプルプログラムでは、名前付きパイプを使って簡単なクライアントサーバーシステムを実装する方法を示しました。ここでは、これまでに学んだ知識をいかしてCDデータベースアプリケーションをクライアントサーバーアプリケーションとして作成し直してみましょう。シグナル処理も組み込み、プロセスが割り込まれた場合の後始末をきちんと行えるようにします。

新しいバージョンについて詳しく見るまえに、アプリケーションをコンパイルしておきましょう。Makefileを使ってserver_appとclient_appを作成してください。server_app -iを実行すると、新しいCDデータベースが初期化されます。いうまでもありませんが、クライアントはサーバーが実行されていなければ起動できません。Makefileを次に示します。

```
all:	server_app client_app

CC=gcc
CFLAGS=-pedantic -Wall

# For debugging un-comment the next line
# DFLAGS=-DDEBUG_TRACE=1 -g

.c.o:
	$(CC) $(CFLAGS) $(DFLAGS) -c $<

app_ui.o: app_ui.c cd_data.h
cd_dbm.o: cd_dbm.c cd_data.h
client_if.o: client_if.c cd_data.h cliserv.h
pipe_imp.o: pipe_imp.c cd_data.h cliserv.h
server.o: server.c cd_data.h cliserv.h

client_app: app_ui.o client_if.o pipe_imp.o
	$(CC) -o client_app  $(DFLAGS) app_ui.o client_if.o pipe_imp.o

server_app: server.o cd_dbm.o pipe_imp.o
	$(CC) -o server_app  $(DFLAGS) server.o cd_dbm.o pipe_imp.o -ldbm
```

11.7.1 目標

このアプリケーションでは、データベースを扱う部分とユーザーインタフェースの部分とを分離することを目標にします。また、サーバープロセスは1つしか実行しませんが、同時に複数のクライアントから接続を受け付けるようにします。既存のコードに加える変更は最小限にとどめ、できるだけこれまでのコードをいかすことを心がけます。

動作環境をシンプルにするために、パイプはアプリケーションの内部で作成（または削除）し、システム管理者が名前付きパイプを作成しなくてもアプリケーションを利用できるようにします。

イベントの発生を待つときに、CPUパワーを消費するビジーウェイト状態を作らないことも重要です。すでに見たように、UNIXではプロセスのブロックが可能なので、リソースをそれほど消費せずにイベントを待つことができます。新しいバージョンでは、ブロックされるパイプの特質をいかしてCPUを効率的に使用します。サーバーは要求が到着するまで、何時間でも待つことができます。

11.7.2 実装

第7章で作成した単一プロセスバージョンのアプリケーションでは、データを操作するための一連のデータアクセスルーチンを用意しました。これらの関数をもう一度リストアップしてみましょう。

```
int database_initialize(const int new_database);
void database_close(void);
cdc_entry get_cdc_entry(const char *cd_catalog_ptr);
cdt_entry get_cdt_entry(const char *cd_catalog_ptr, const int track_no);
int add_cdc_entry(const cdc_entry entry_to_add);
int add_cdt_entry(const cdt_entry entry_to_add);
int del_cdc_entry(const char *cd_catalog_ptr);
int del_cdt_entry(const char *cd_catalog_ptr, const int track_no);
cdc_entry search_cdc_entry(const char *cd_catalog_ptr,
                           int *first_call_ptr);
```

これらの関数は、クライアントとサーバーを明確に分離するのに利用できそうです。

単一プロセス方式の実装では、コンパイル後のプログラムは1つですが、次の図のように2つの部分を持つアプリケーションと考えることができます。

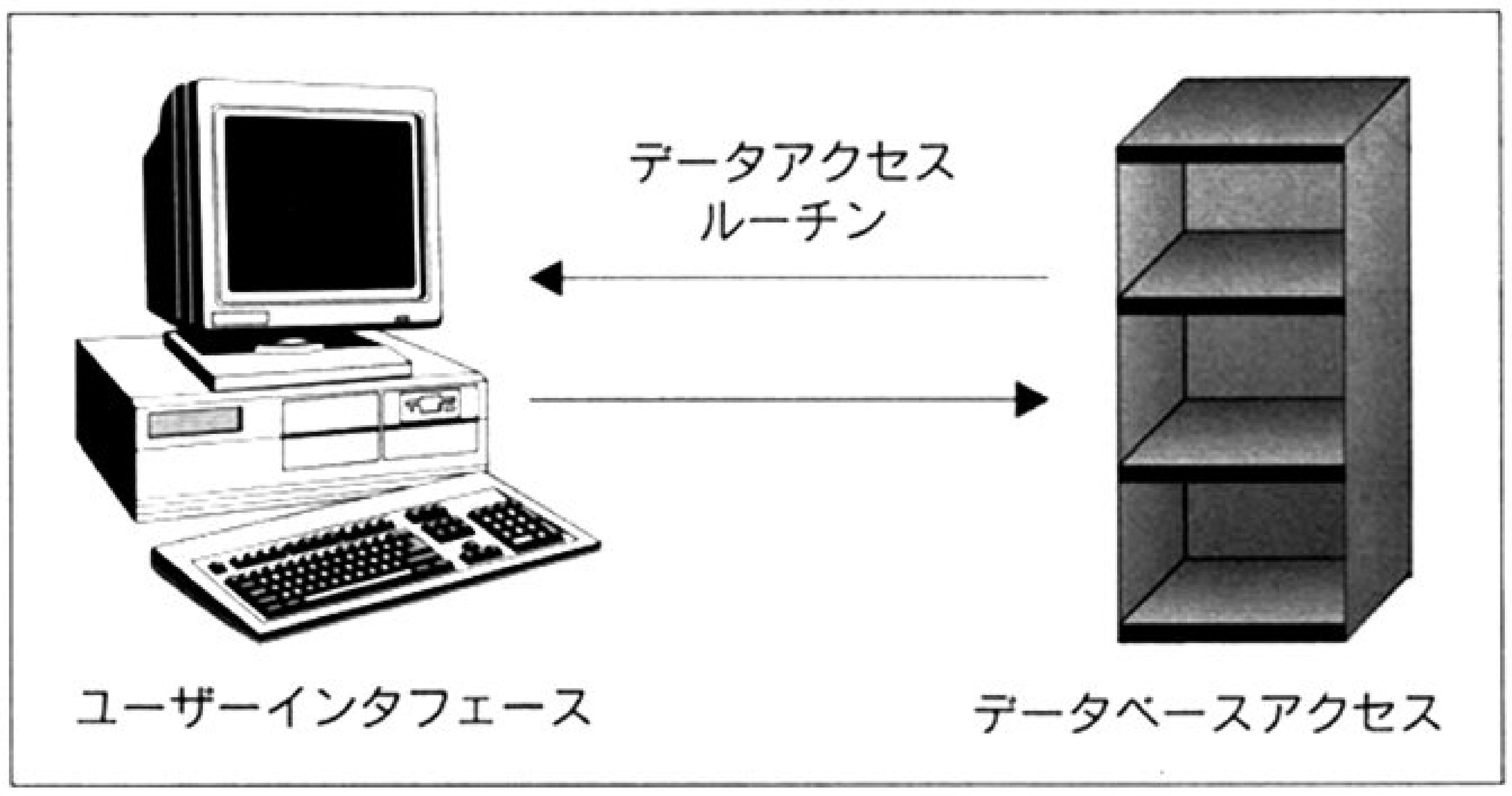


図11.6 単一プロセス方式の実装

クライアントサーバー方式の実装では、これら2つの部分の間に名前付きパイプとその他の必要なコードを挿入すればよいことになります。これを図で示すと、次のようになります。

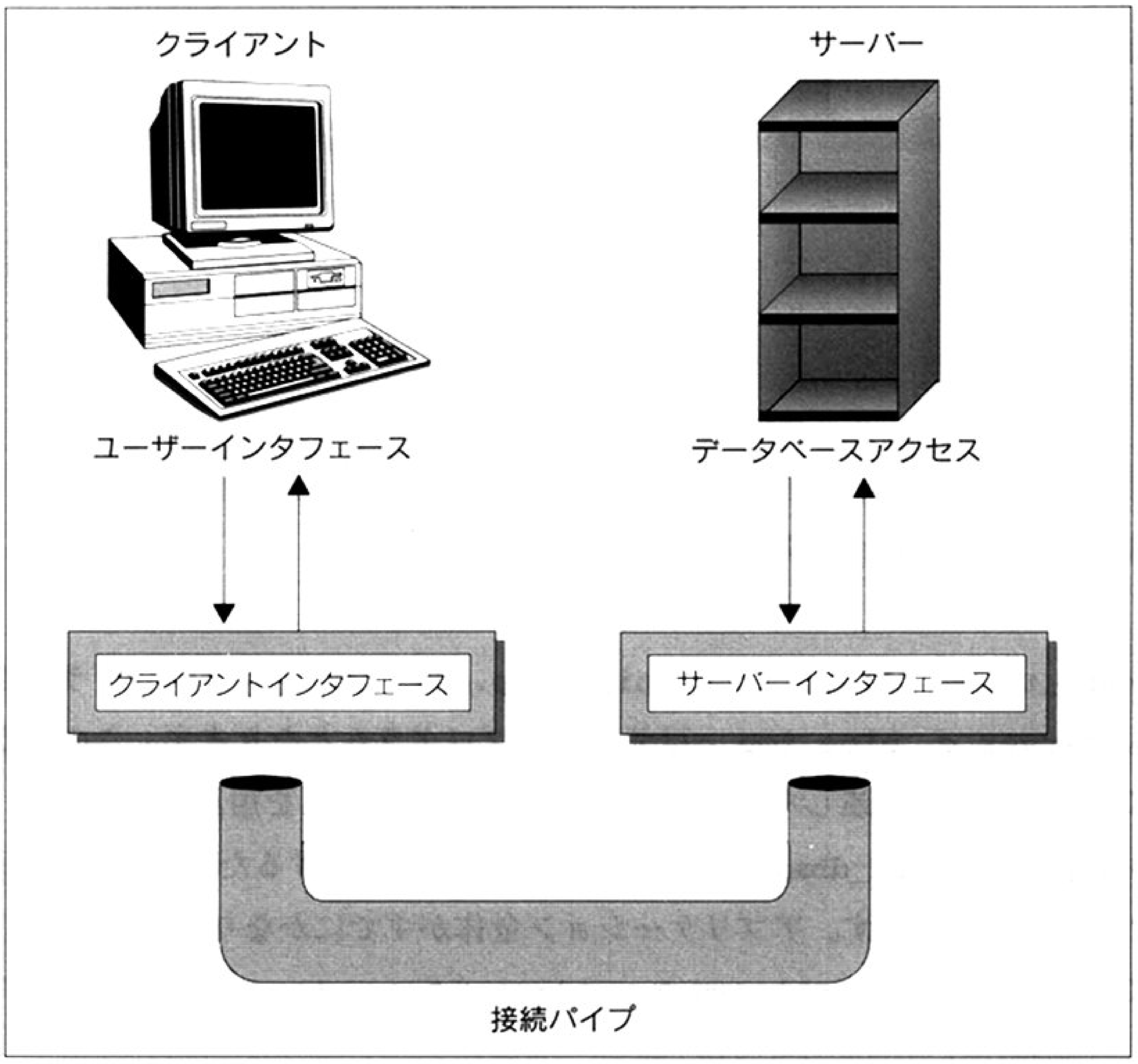


図11.7 クライアントサーバー方式の実装

新しいバージョンでは、クライアントインタフェースとサーバーインタフェースの両方のルーチンを1つのファイル**pipe_imp.c**に記述することにします。これで、クライアントサーバー方式の名前付きパイプに依存するすべてのコードを1つのファイルにまとめておくことができます。パイプを介してやりとりするデータの形式とパッケージングは、名前付きパイプを実装するルー

チンとは分けておきます。ソースファイルの数は増えますが、論理的にはよりすっきりした形になります。アプリケーション内部での呼び出しの構造は、次の図に示すとおりです。

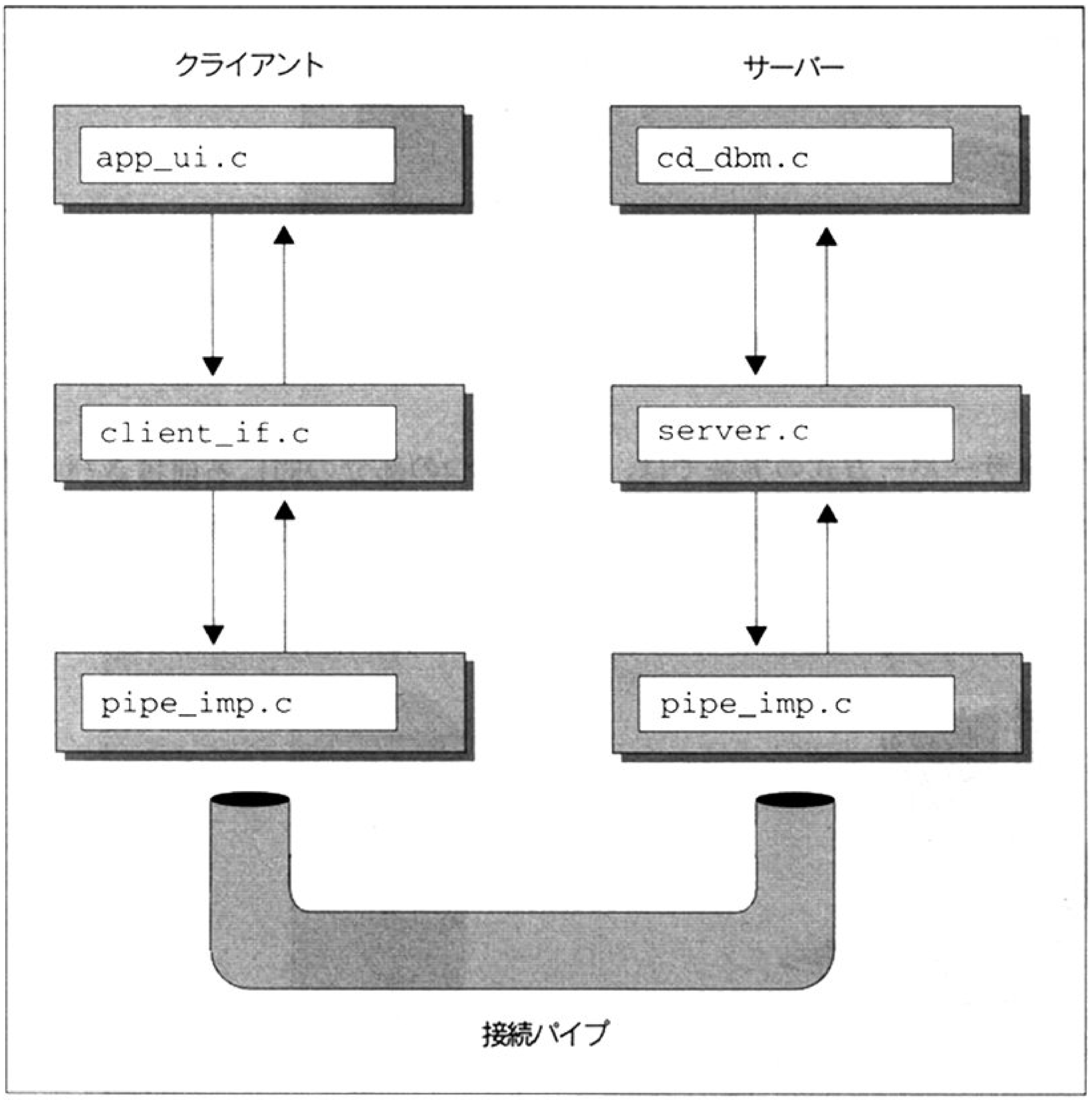


図11.8 アプリケーション内部の呼び出しの構造

ファイル**app_ui.c**、**client_if.c**、**pipe_imp.c**はクライアントプログラムを構成します。**cd_dbm.c**、**server.c**、**pipe_imp.c**はサーバープログラムとなります。このほかに、2つのプログラムに共通の定義を記述したヘッダーファイル**cliserv.h**を用意します。

ファイル**app_ui.c**と**cd_dbm.c**は、プログラムを2つに分けるための修正点を除けば、既存のコードとほとんど同じです。アプリケーション全体がすでにかなり大きくなっているので、ここでは主にヘッダーファイル**cliserv.h**、**client_if.c**、**pipe_imp.c**のコードについて見ていきます。

最初に示すのは**cliserv.h**です。このファイルでは、クライアントサーバーインタフェースを定義します。このヘッダーファイルは、クライアントプログラムとサーバープログラムの両方でインクルードする必要があります。

メモ

このファイルの一部は、特定のクライアントサーバーモデルの実装（この章で作成するバージョンでは名前付きパイプ）に依存しています。次の章の最後では、また別のクライアントサーバーモデルに基づく実装を取り上げます。

例題 ヘッダーファイルcliserv.h

1 まず、機能テストマクロ`_POSIX_SOURCE`を定義し、コードのコンパイルが実装に依存する独自の制限ではなく、POSIXの定義を使って行われるようにします。これは、移植性を高めるためです。

```
#define _POSIX_SOURCE
```

2 必要なヘッダーファイルをインクルードします。

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>
```

3 名前付きパイプを定義します。名前付きパイプはサーバー用に1つ、各クライアント用に1つ用意します。クライアント用の名前付きパイプでは、クライアントのプロセスIDを名前に埋め込み、パイプが一意になるようにします。

```
#define SERVER_PIPE "/tmp/server_pipe"
#define CLIENT_PIPE "/tmp/client_%d_pipe"

#define ERR_TEXT_LEN 80
```

4 コマンドは`#define`ではなく、列挙型で実装します。

メモ

列挙型を使うと、コンパイラによる型チェックが可能になります。また、多くのデバッガでは、`#define`によって定義された名前は表示できませんが、列挙型定数の名前は表示できるので、アプリケーションのデバッグがやりやすくなります。

最初の`typedef`はサーバーに送られる要求の種類の定義で、2番目の`typedef`はサーバーからクライアントへの応答の定義です。

```
typedef enum {
    s_create_new_database = 0,
    s_get_cdc_entry,
    s_get_cdt_entry,
    s_add_cdc_entry,
    s_add_cdt_entry,
    s_del_cdc_entry,
    s_del_cdt_entry,
    s_find_cdc_entry
} client_request_e;

typedef enum {
    r_success = 0,
    r_failure,
    r_find_no_more
} server_response_e;
```

5 2つのプロセス間でやりとりされるメッセージの構造を宣言します。

注意

> 実際には、1つの応答の中でcdc_entryとcdt_entryの両方を返す必要はないので、これらをunionでまとめることも可能です。ここでは、複雑になるのを避けるためにcdc_entryとcdt_entryを独立したメンバとして用意します。また、このほうがコードの保守も容易です。

```
typedef struct {
    pid_t               client_pid;
    client_request_e    request;
    server_response_e   response;
    cdc_entry           cdc_entry_data;
    cdt_entry           cdt_entry_data;
    char                error_text[ERR_TEXT_LEN + 1];
} message_db_t;
```

6 最後に、データ転送を実行するパイプインタフェース関数の宣言を記述します。これらの関数はpipe_imp.cで実装します。宣言は2つのブロックに分かれていますが、最初のものがサーバー側の関数、2番目のものがクライアント側の関数です。

```
int server_starting(void);
void server_ending(void);
int read_request_from_client(message_db_t *rec_ptr);
int start_resp_to_client(const message_db_t mess_to_send);
int send_resp_to_client(const message_db_t mess_to_send);
void end_resp_to_client(void);

int client_starting(void);
void client_ending(void);
int send_mess_to_server(message_db_t mess_to_send);
int start_resp_from_server(void);
```

```
int read_resp_from_server(message_db_t *rec_ptr);
void end_resp_from_server(void);
```

以下では、クライアントインタフェース関数と、pipe_imp.cで実装するサーバー側関数およびクライアント側関数について説明します。また、必要に応じてほかのソースコードも参照します。

11.7.3 クライアントインタフェース

client_if.cではデータベースアクセスルーチンを提供します。これらの関数は、実際にはmessage_db_t構造体に要求を埋め込み、pipe_imp.cのルーチンを使ってサーバーに要求を転送します。ただし、少なくとも表面上は以前のデータベースアクセスルーチンと変わらないので、もとのapp_ui.cに対する大幅な変更は不要です。

例題 クライアントのインタープリタclient_if.c

1 このファイルでは、cd_data.hでプロトタイプ宣言されている9つのデータベース関数を実装します。これらの関数は仲介役として機能し、サーバーに要求を渡すとともに、サーバーからの応答を返します。

まず、ヘッダーファイルをインクルードします。

```
#define _POSIX_SOURCE

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>

#include "cd_data.h"
#include "cliserv.h"
```

2 static変数mypidは、getpidの呼び出し回数をできるだけ減らすために用意します。ローカル関数read_one_responseは、重複するコードを1つにまとめるための関数です。

```
static pid_t mypid;

static int read_one_response(message_db_t *rec_ptr);
```

3 database_initialize関数とdatabase_close関数は、このバージョンではパイプインタフェースのクライアント側の初期化と、クライアント終了時の不要な名前付きパイプの削除を行います。

```
int database_initialize(const int new_database)
{
    if (!client_starting()) return(0);
    mypid = getpid();
    return(1);
}

void database_close(void) {
    client_ending();
}
```

4 get_cdc_entryは、指定されたCDカタログタイトルのカタログエントリをデータベースから取得します。この関数では、要求をmessage_db_t構造体に埋め込んでサーバーに渡します。返された応答は、別のmessage_db_t構造体に格納します。見つかったエントリはmessage_db_t構造体のメンバcdc_entryに格納されるので、これを関数の戻り値とします。

```
cdc_entry get_cdc_entry(const char *cd_catalog_ptr)
{
    cdc_entry ret_val;
    message_db_t mess_send;
    message_db_t mess_ret;

    ret_val.catalog[0] = '\0';
    mess_send.client_pid = mypid;
    mess_send.request = s_get_cdc_entry;
    strcpy(mess_send.cdc_entry_data.catalog, cd_catalog_ptr);

    if (send_mess_to_server(mess_send)) {
        if (read_one_response(&mess_ret)) {
            if (mess_ret.response == r_success) {
                ret_val = mess_ret.cdc_entry_data;
            } else {
                fprintf(stderr, mess_ret.error_text);
            }
        } else {
            fprintf(stderr, "Server failed to respond\n");
        }
    } else {
        fprintf(stderr, "Server not accepting requests\n");
    }
    return(ret_val);
}
```

5 次は、コードの重複を避けるために使うread_one_response関数です。

```
static int read_one_response(message_db_t *rec_ptr)
{
    int return_code = 0;
    if (!rec_ptr) return(0);

    if (start_resp_from_server()) {
        if (read_resp_from_server(rec_ptr)) {
            return_code = 1;
        }
        end_resp_from_server();
    }
    return(return_code);
}
```

6 get_cdc_entry関数と同じ要領で、get_xxx、del_xxx、add_xxxといったルーチンを用意します。まず、CDトラックを取得する関数です。

```
cdt_entry get_cdt_entry(const char *cd_catalog_ptr, const int track_no)
{
    cdt_entry ret_val;
    message_db_t mess_send;
    message_db_t mess_ret;

    ret_val.catalog[0] = '\0';
    mess_send.client_pid = mypid;
    mess_send.request = s_get_cdt_entry;
    strcpy(mess_send.cdt_entry_data.catalog, cd_catalog_ptr);
    mess_send.cdt_entry_data.track_no = track_no;

    if (send_mess_to_server(mess_send)) {
        if (read_one_response(&mess_ret)) {
            if (mess_ret.response == r_success) {
                ret_val = mess_ret.cdt_entry_data;
            } else {
                fprintf(stderr, mess_ret.error_text);
            }
        } else {
            fprintf(stderr, "Server failed to respond\n");
        }
    } else {
        fprintf(stderr, "Server not accepting requests\n");
    }
    return(ret_val);
}
```

7 次はカタログにデータを追加する関数と、トラックデータベースにデータを追加する関数です。

```
int add_cdc_entry(const cdc_entry entry_to_add)
{
    message_db_t mess_send;
    message_db_t mess_ret;

    mess_send.client_pid = mypid;
    mess_send.request = s_add_cdc_entry;
    mess_send.cdc_entry_data = entry_to_add;

    if (send_mess_to_server(mess_send)) {
        if (read_one_response(&mess_ret)) {
            if (mess_ret.response == r_success) {
                return(1);
            } else {
                fprintf(stderr, mess_ret.error_text);
            }
        } else {
            fprintf(stderr, "Server failed to respond\n");
        }
    } else {
        fprintf(stderr, "Server not accepting requests\n");
    }
    return(0);
}

int add_cdt_entry(const cdt_entry entry_to_add)
{
    message_db_t mess_send;
    message_db_t mess_ret;

    mess_send.client_pid = mypid;
    mess_send.request = s_add_cdt_entry;
    mess_send.cdt_entry_data = entry_to_add;

    if (send_mess_to_server(mess_send)) {
        if (read_one_response(&mess_ret)) {
            if (mess_ret.response == r_success) {
                return(1);
            } else {
                fprintf(stderr, mess_ret.error_text);
            }
        } else {
            fprintf(stderr, "Server failed to respond\n");
        }
    } else {
        fprintf(stderr, "Server not accepting requests\n");
    }
    return(0);
}
```

8 最後の2つはデータを削除する関数です。

```
int del_cdc_entry(const char *cd_catalog_ptr)
{
    message_db_t mess_send;
    message_db_t mess_ret;

    mess_send.client_pid = mypid;
    mess_send.request = s_del_cdc_entry;
    strcpy(mess_send.cdc_entry_data.catalog, cd_catalog_ptr);

    if (send_mess_to_server(mess_send)) {
        if (read_one_response(&mess_ret)) {
            if (mess_ret.response == r_success) {
                return(1);
            } else {
                fprintf(stderr, mess_ret.error_text);
            }
        } else {
            fprintf(stderr, "Server failed to respond\n");
        }
    } else {
        fprintf(stderr, "Server not accepting requests\n");
    }
    return(0);
}

int del_cdt_entry(const char *cd_catalog_ptr, const int track_no)
{
    message_db_t mess_send;
    message_db_t mess_ret;

    mess_send.client_pid = mypid;
    mess_send.request = s_del_cdt_entry;
    strcpy(mess_send.cdt_entry_data.catalog, cd_catalog_ptr);
    mess_send.cdt_entry_data.track_no = track_no;

    if (send_mess_to_server(mess_send)) {
        if (read_one_response(&mess_ret)) {
            if (mess_ret.response == r_success) {
                return(1);
            } else {
                fprintf(stderr, mess_ret.error_text);
            }
        } else {
            fprintf(stderr, "Server failed to respond\n");
        }
    } else {
        fprintf(stderr, "Server not accepting requests\n");
    }
    return(0);
}
```

◆ データベースの検索

CDキーを検索する関数はやや複雑になります。すでに第7章で説明したように、最初にこの関数を呼び出すときには*first_call_ptrに真を設定します。この場合には、最初に見つかったエントリが返されます。以後の呼び出しでは*first_call_ptrが偽になり、呼び出されるたびに見つかったエントリが返されます。

しかし、新しいバージョンではアプリケーションを2つのプロセスに分割しているので、サーバー側でエントリを1つずつ処理するわけにはいきません。なぜなら、サーバーで検索を実行しているときに、ほかのクライアントから別の検索要求が送られてくる可能性があるからです。サーバー側では、各クライアントのコンテキスト（どこまで検索を実行したか）を保持しておくことはできません。クライアント側では、目的のCDが見つかった時点で検索を中止するかもしれませんし、場合によっては異常終了することもあるからです。

問題を回避するには、検索のやり方自体を変更する方法と、処理手順の面倒な側面をインタフェースルーチンの中に隠してしまう方法とがあります。ここでは、後者の方法を採用し、サーバーから一致するすべてのエントリを受け取り、これを一時ファイルに格納してクライアントからの要求に備えることにします。

例題 検索

1 この関数は、send_mess_to_server、start_resp_from_server、read_resp_from_serverという3つのパイプ関数を呼び出しているので一見複雑そうに見えますが、実際にはそれほどではありません。順を追って見ていきましょう。

```
cdc_entry search_cdc_entry(const char *cd_catalog_ptr, int *first_call_ptr)
{
    message_db_t mess_send;
    message_db_t mess_ret;

    static FILE *work_file = (FILE *)0;
    static int entries_matching = 0;
    cdc_entry ret_val;

    ret_val.catalog[0] = '\0';

    if (!work_file && (*first_call_ptr == 0)) return(ret_val);
```

2 最初は、検索開始時、つまり*first_call_ptrに真が設定されている場合の処理です。まず、*first_call_ptrに偽を設定します。次にwork_fileを作成し、クライアントのメッセージ構造体を初期化します。

```
    if (*first_call_ptr) {
        *first_call_ptr = 0;
        if (work_file) fclose(work_file);
```

```
    work_file = tmpfile();
    if (!work_file) return(ret_val);

    mess_send.client_pid = mypid;
    mess_send.request = s_find_cdc_entry;
    strcpy(mess_send.cdc_entry_data.catalog, cd_catalog_ptr);
```

3 3段階にネストした条件判定を行います。ここでは、pipe_imp.cで実装している関数を呼び出します。サーバーにメッセージが正常に送信された場合、クライアントはサーバーからの応答を待ちます。サーバーからの読み取りが成功すると一致するエントリが返されるので、これをwork_fileに格納し、entries_matchingカウンタをインクリメントします。

```
    if (send_mess_to_server(mess_send)) {
        if (start_resp_from_server()) {
            while (read_resp_from_server(&mess_ret)) {
                if (mess_ret.response == r_success) {
                    fwrite(&mess_ret.cdc_entry_data,
                            sizeof(cdc_entry), 1, work_file);
                    entries_matching++;
                } else {
                    break;
                }
            } /* while */
        } else {
            fprintf(stderr, "Server not responding\n");
        }
    } else {
        fprintf(stderr, "Server not accepting requests\n");
    }
```

4 次に、一致するエントリがあったかどうかを調べ、work_file内の次にデータを書き込む位置にシークします。

```
    if (entries_matching == 0) {
        fclose(work_file);
        work_file = (FILE *)0;
        return(ret_val);
    }
    (void)fseek(work_file, 0L, SEEK_SET);
```

5 検索開始時の呼び出しでなければ、一致するエントリが残っているかどうかをチェックします。残っている場合には、該当するエントリをret_val構造体に読み込みます。

```
  } else {
    /* not *first_call_ptr */
    if (entries_matching == 0) {
        fclose(work_file);
        work_file = (FILE *)0;
```

11

```
            return(ret_val);
        }
    }

    fread(&ret_val, sizeof(cdc_entry), 1, work_file);
    entries_matching--;

    return(ret_val);
}
```

11.7.4 サーバーインタフェース

クライアント側にはapp_ui.cプログラムに対するインタフェースがありますが、同様にサーバー側でもcd_dbm.c（第7章のバージョンのcd_access.cに相当）を制御するためのプログラムを用意します。

例題 server.c

1 これまで同様、マクロ_POSIX_SOURCEを定義し、ヘッダーファイルをインクルードします。また、いくつかのグローバル変数とprocess_command関数のプロトタイプを宣言し、正常に終了するためのcatch_signals関数を用意します。

```
#define _POSIX_SOURCE

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <limits.h>
#include <signal.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>

#include "cd_data.h"
#include "cliserv.h"

int save_errno;
static int server_running = 1;

static void process_command(const message_db_t mess_command);

void catch_signals()
{
    server_running = 0;
}
```

2 次はmain関数です。シグナル捕捉ルーチンが正しく設定されていることをチェックしたあと、コマンドラインで-iが指定されているかどうかを調べます。-iが指定されていた場合には、新しいデータベースを作成します。cd_dbm.cで実装されているdatabase_initializeの呼び出しが失敗した場合には、エラーメッセージを表示します。すべての処理が問題なく終了し、サーバーが実行されている場合には、クライアントからの要求をprocess_command関数に渡します。

```
int main(int argc, char *argv[]) {
    struct sigaction new_action, old_action;
    message_db_t mess_command;
    int database_init_type = 0;

    new_action.sa_handler = catch_signals;
    sigemptyset(&new_action.sa_mask);
    new_action.sa_flags = 0;
    if ((sigaction(SIGINT, &new_action, &old_action) != 0) ||
        (sigaction(SIGHUP, &new_action, &old_action) != 0) ||
        (sigaction(SIGTERM, &new_action, &old_action) != 0)) {
        fprintf(stderr, "Server startup error, signal catching failed\n");
        exit(EXIT_FAILURE);
    }

    if (argc > 1) {
        argv++;
        if (strncmp("-i", *argv, 2) == 0) database_init_type = 1;
    }
    if (!database_initialize(database_init_type)) {
                fprintf(stderr, "Server error:-\
                         could not initialize database\n");
                exit(EXIT_FAILURE);
    }

    if (!server_starting()) exit(EXIT_FAILURE);

    while(server_running) {
        if (read_request_from_client(&mess_command)) {
            process_command(mess_command);
        } else {
            if(server_running) fprintf(stderr, "Server ended - can not \
                                            read pipe\n");
            server_running = 0;
        }
    } /* while */
    server_ending();
    exit(EXIT_SUCCESS);
}
```

3 クライアントからのメッセージは、このprocess_command関数に渡されます。caseステートメントによってcd_dbm.c内の適切な関数を呼び出します。

```
static void process_command(const message_db_t comm)
{
    message_db_t resp;
    int first_time = 1;

    resp = comm; /* copy command back, then change resp as required */

    if (!start_resp_to_client(resp)) {
        fprintf(stderr, "Server Warning:-\
                 start_resp_to_client %d failed\n", resp.client_pid);
        return;
    }

    resp.response = r_success;
    memset(resp.error_text, '\0', sizeof(resp.error_text));
    save_errno = 0;

    switch(resp.request) {
        case s_create_new_database:
            if (!database_initialize(1)) resp.response = r_failure;
            break;
        case s_get_cdc_entry:
            resp.cdc_entry_data =
                           get_cdc_entry(comm.cdc_entry_data.catalog);
            break;
        case s_get_cdt_entry:
            resp.cdt_entry_data =
                           get_cdt_entry(comm.cdt_entry_data.catalog,
                                         comm.cdt_entry_data.track_no);
            break;
        case s_add_cdc_entry:
            if (!add_cdc_entry(comm.cdc_entry_data)) resp.response =
                           r_failure;
            break;
        case s_add_cdt_entry:
            if (!add_cdt_entry(comm.cdt_entry_data)) resp.response =
                           r_failure;
            break;
        case s_del_cdc_entry:
            if (!del_cdc_entry(comm.cdc_entry_data.catalog)) resp.response
                         = r_failure;
            break;
        case s_del_cdt_entry:
            if (!del_cdt_entry(comm.cdt_entry_data.catalog,
                 comm.cdt_entry_data.track_no)) resp.response = r_failure;
            break;
        case s_find_cdc_entry:
            do {
                resp.cdc_entry_data =
                          search_cdc_entry(comm.cdc_entry_data.catalog,
                                           &first_time);
                if (resp.cdc_entry_data.catalog[0] != 0) {
                    resp.response = r_success;
```

```
                    if (!send_resp_to_client(resp)) {
                        fprintf(stderr, "Server Warning:-\
                            failed to respond to %d\n", resp.client_pid);
                        break;
                    }
                } else {
                    resp.response = r_find_no_more;
                }
            } while (resp.response == r_success);
        break;
        default:
            resp.response = r_failure;
            break;
    } /* switch */

    sprintf(resp.error_text, "Command failed:\n\t%s\n",
             strerror(save_errno));

    if (!send_resp_to_client(resp)) {
        fprintf(stderr, "Server Warning:-\
                failed to respond to %d\n", resp.client_pid);
    }

    end_resp_to_client();
    return;
}
```

パイプの実装の説明に進む前に、クライアントプロセスとサーバープロセスの間でデータをやりとりする際のイベントの順序について見ておきましょう。次の図は、クライアントプロセスとサーバープロセスの起動、およびコマンドや応答の処理の流れを表したものです。

実際の実装では、検索要求に関する部分がやや複雑になっています。クライアントがサーバーに渡すコマンドは1つですが、サーバーからは1つまたは複数の応答が返されることを期待しているからです。

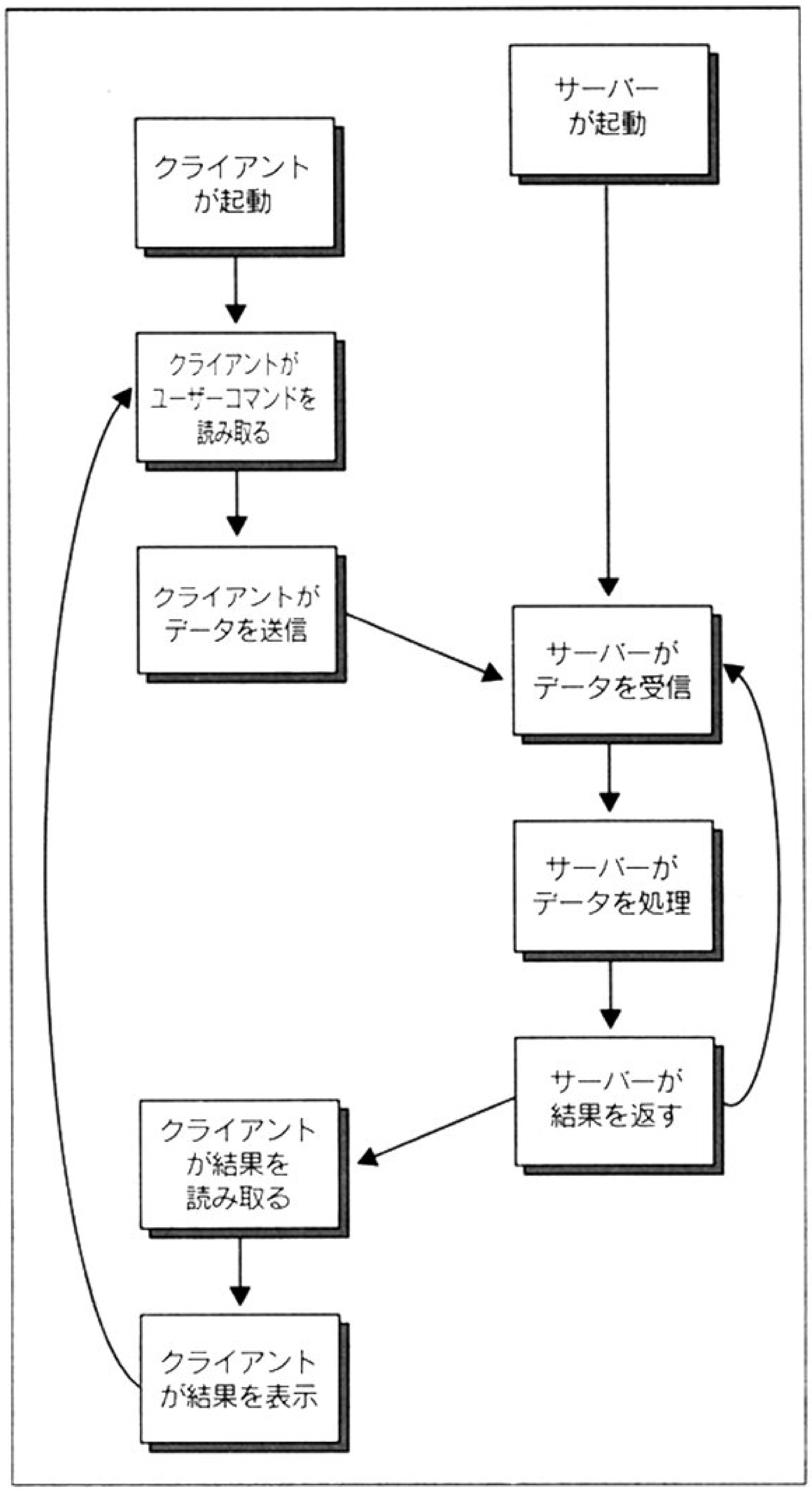


図11.9 プロセス間のイベントの順序

11.7.5 パイプ

ここでは、パイプを実装するファイル**pipe_imp.c**について詳しく見ていきます。このファイルには、クライアント側の関数とサーバー側の関数の両方が含まれています。

> **メモ**
> シンボル**DEBUG_TRACE**を定義すると、クライアントプロセスとサーバープロセスが互いにメッセージをやりとりするときの関数呼び出しの手順を表示することができます。

例題 pipe_imp.cのヘッダー部分

1 ヘッダーファイルをインクルードします。

```
#include "cd_data.h"
#include "cliserv.h"
```

2 ファイル内のさまざまな関数で使う変数を定義し、初期化します。

```
static int server_fd = -1;
static pid_t mypid = 0;
static char client_pipe_name[PATH_MAX + 1] = {'\0'};
static int client_fd = -1;
static int client_write_fd = -1;
```

◆ サーバー側関数

まず、サーバー側の関数を取り上げます。最初に示すのは、名前付きパイプのオープンとクローズ、およびクライアントからのメッセージの読み取りを行う関数です。そのあとで、クライアントからのメッセージに埋め込まれたプロセスIDに基づいてクライアントパイプをオープンし、メッセージの送信とパイプのクローズを行う関数を示します。

例題 サーバー関数

1 server_starting関数は、コマンドを読み取るための名前付きパイプを作成します。名前付きパイプを作成したら、読み取り用にオープンします。このopenは、クライアントが書き込み用にパイプをオープンするまでブロックされます。ブロックされるモードを使用するのは、クライアントからコマンドが送信されてくるのを待ちながら、パイプに対してブロックされるreadを実行できるようにするためです。

```
int server_starting(void)
{
    #if DEBUG_TRACE
        printf("%d :- server_starting()\n",  getpid());
    #endif

    (void)unlink(SERVER_PIPE);
    if (mkfifo(SERVER_PIPE, 0777) == -1) {
        fprintf(stderr, "Server startup error, no FIFO created\n");
        return(0);
    }

    if ((server_fd = open(SERVER_PIPE, O_RDONLY)) == -1) {
        if (errno == EINTR) return(0);
```

```
        fprintf(stderr, "Server startup error, no FIFO opened\n");
        return(0);
    }
    return(1);
}
```

2 サーバーを終了するときには、名前付きパイプを削除し、サーバーが実行されていないことをクライアントが検出できるようにします。

```
void server_ending(void)
{
    #if DEBUG_TRACE
        printf("%d :- server_ending()\n",  getpid());
    #endif

    (void)close(server_fd);
    (void)unlink(SERVER_PIPE);
}
```

3 次のread_request_from_client関数では、サーバーパイプに対してブロックされる読み取りを実行します。この操作は、クライアントがメッセージを書き込むまでブロックされます。

```
int read_request_from_client(message_db_t *rec_ptr)
{
    int return_code = 0;
    int read_bytes;

    #if DEBUG_TRACE
        printf("%d :- read_request_from_client()\n",  getpid());
    #endif

    if (server_fd != -1) {
        read_bytes = read(server_fd, rec_ptr, sizeof(*rec_ptr));
...
    }
    return(return_code);
}
```

4 書き込み用にパイプをオープンしているクライアントがない場合、readは0を返します（EOFを検出します）。この場合、サーバーはパイプをいったんクローズして再度オープンし、クライアントがパイプをオープンするまでブロックされるようにします。これはサーバーが起動したときの動作と同じで、サーバーを再初期化することになります。このコードは、上の関数の...の部分に挿入します。

```
        if (read_bytes == 0) {
            (void)close(server_fd);
```

```
            if ((server_fd = open(SERVER_PIPE, O_RDONLY)) == -1) {
                if (errno != EINTR) {
                    fprintf(stderr, "Server error, FIFO open failed\n");
                }
                return(0);
            }
            read_bytes = read(server_fd, rec_ptr, sizeof(*rec_ptr));
        }
        if (read_bytes == sizeof(*rec_ptr)) return_code = 1;
```

サーバーは、複数のクライアントに同時にサービスを提供する単一のプロセスです。各クライアントは応答を受け取るのにそれぞれ独自のパイプを使うので、サーバー側でもクライアントごとに異なるパイプに書き込みを行う必要があります。ファイルデスクリプタは限りのあるリソースなので、サーバーでは送信するデータがある場合に限ってクライアントパイプをオープンします。

クライアントパイプのオープン、書き込み、クローズは、それぞれ独立した関数で実装します。検索要求に対しては複数の結果を返すことがありますが、このような場合でも関数を独立させておけば、オープンとクローズは1回だけ、書き込みは複数回といった処理が可能になります。

例題 パイプの接続

1 まず、クライアントパイプをオープンします。

```
int start_resp_to_client(const message_db_t mess_to_send)
{
    #if DEBUG_TRACE
        printf("%d :- start_resp_to_client()\n", getpid());
    #endif

    (void)sprintf(client_pipe_name, CLIENT_PIPE, mess_to_send.client_pid);
    if ((client_fd = open(client_pipe_name, O_WRONLY)) == -1) return(0);
    return(1);
}
```

2 メッセージの送信には、次のsend_resp_to_client関数を使います。送信されたメッセージを受け取るクライアント側の関数については、あとで示します。

```
int send_resp_to_client(const message_db_t mess_to_send)
{
    int write_bytes;

    #if DEBUG_TRACE
        printf("%d :- send_resp_to_client()\n", getpid());
    #endif

    if (client_fd == -1) return(0);
```

```
    write_bytes = write(client_fd, &mess_to_send, sizeof(mess_to_send));
    if (write_bytes != sizeof(mess_to_send)) return(0);
    return(1);
}
```

3 クライアントパイプをクローズします。

```
void end_resp_to_client(void)
{
    #if DEBUG_TRACE
        printf("%d :- end_resp_to_client()\n",  getpid());
    #endif

    if (client_fd != -1) {
        (void)close(client_fd);
        client_fd = -1;
    }
}
```

◆ クライアント側関数

クライアント側関数は、サーバー側関数と呼応する関数群です。

例題 クライアント関数

1 client_starting関数は、サーバーにアクセス可能であることを確認し、クライアント側パイプを初期化します。

```
int client_starting(void)
{
    #if DEBUG_TRACE
        printf("%d :- client_starting\n",  getpid());
    #endif

    mypid = getpid();
    if ((server_fd = open(SERVER_PIPE, O_WRONLY)) == -1) {
        fprintf(stderr, "Server not running\n");
        return(0);
    }

    (void)sprintf(client_pipe_name, CLIENT_PIPE, mypid);
    (void)unlink(client_pipe_name);
    if (mkfifo(client_pipe_name, 0777) == -1) {
        fprintf(stderr, "Unable to create client pipe %s\n",
                    client_pipe_name);
        return(0);
    }
    return(1);
}
```

2 client_ending関数は、ファイルデスクリプタをクローズし、不要になった名前付きパイプを削除します。

```
void client_ending(void)
{
    #if DEBUG_TRACE
        printf("%d :- client_ending()\n",  getpid());
    #endif

    if (client_write_fd != -1) (void)close(client_write_fd);
    if (client_fd != -1) (void)close(client_fd);
    if (server_fd != -1) (void)close(server_fd);
    (void)unlink(client_pipe_name);
}
```

3 send_mess_to_server関数は、サーバーパイプを介して要求を送信します。

```
int send_mess_to_server(message_db_t mess_to_send)
{
    int write_bytes;

    #if DEBUG_TRACE
        printf("%d :- send_mess_to_server()\n",  getpid());
    #endif

    if (server_fd == -1) return(0);
    mess_to_send.client_pid = mypid;
    write_bytes = write(server_fd, &mess_to_send, sizeof(mess_to_send));
    if (write_bytes != sizeof(mess_to_send)) return(0);
    return(1);
}
```


前に示したサーバー側関数の場合と同様、クライアントも3つの関数を使ってサーバーから結果を受け取ります。

例題 サーバーからの結果の取得

1 クライアント関数start_resp_from_serverは、サーバーからの応答の受信を開始します。クライアントパイプを読み取り専用でオープンし、次に同じパイプを書き込み専用でオープンします。このようにする理由についてはあとで説明します。

```
int start_resp_from_server(void)
{
    #if DEBUG_TRACE
        printf("%d :- start_resp_from_server()\n",  getpid());
    #endif

    if (client_pipe_name[0] == '\0') return(0);
```

```
    if (client_fd != -1) return(1);

    client_fd = open(client_pipe_name, O_RDONLY);
    if (client_fd != -1) {
        client_write_fd = open(client_pipe_name, O_WRONLY);
        if (client_write_fd != -1) return(1);
        (void)close(client_fd);
        client_fd = -1;
    }
    return(0);
}
```

2 次のread_resp_from_serverは、サーバーからの応答を読み取り、一致するデータベースエントリを取得する関数です。

```
int read_resp_from_server(message_db_t *rec_ptr)
{
    int read_bytes;
    int return_code = 0;

    #if DEBUG_TRACE
        printf("%d :- read_resp_from_server()\n",  getpid());
    #endif

    if (!rec_ptr) return(0);
    if (client_fd == -1) return(0);

    read_bytes = read(client_fd, rec_ptr, sizeof(*rec_ptr));
    if (read_bytes == sizeof(*rec_ptr)) return_code = 1;
    return(return_code);
}
```

3 最後は、サーバーからの応答の終わりに実行するクライアント関数です。

```
void end_resp_from_server(void)
{
    #if DEBUG_TRACE
        printf("%d :- end_resp_from_server()\n",  getpid());
    #endif

    /* This function is empty in the pipe implementation */
}
```

解説

start_resp_from_server関数では、読み取り用にクライアントパイプをオープンしたあと、次のコードで同じパイプを書き込み用にオープンしています。

```
client_write_fd = open(client_pipe_name, O_WRONLY);
```

これは、短い間に続けて送られてきたクライアントの複数の要求にサーバーが応答する際、レースコンディションが発生しないようにするためです。

次のようなケースを想定してみてください。

1. クライアントがサーバーに要求を送る。
2. サーバーは送られてきた要求を読み取り、クライアントパイプをオープンして応答を返すが、クライアントパイプを閉じるまで一時停止する。
3. クライアントが読み取り用にパイプをオープンし、最初の応答を読み取ってパイプをクローズする。
4. クライアントが新しいコマンドを送信し、読み取り用にクライアントパイプをオープンする。
5. サーバーが実行を再開し、クライアントパイプのサーバー側をクローズする。

さて、クライアントはこのとき、次の要求に対する応答を取得するためにパイプを読み取ろうとしますが、書き込み用にクライアントパイプをオープンしているプロセスが存在しないので、**read**は0バイトを返します。これでは、目的の応答を読み取ることができません。

しかし、クライアントが読み取り用と書き込み用の両方でパイプをオープンすれば、パイプを繰り返しオープンする必要はなくなり、レースコンディションを回避することができます。クライアント側では決してパイプへの書き込みを行わないので、読み取るデータが不正になる危険性もありません。

11.7.6 新しいバージョンのまとめ

新しいバージョンでは、CDデータベースアプリケーションをクライアントとサーバーに分離しました。2つの部分に分けたことにより、ユーザーインタフェースとデータベース操作の部分をそれぞれ独立して開発することが可能になります。また、データベースインタフェースを明確に定義することで、アプリケーションの各部がコンピュータリソースを有効に活用できることがわかります。この方法をさらに進めてパイプの代わりにネットワークを介したデータのやりとりを行うようにすれば、専用のデータベースサーバーマシンを使うこともできます。ネットワークを介したデータのやりとりについては、第13章で取り上げます。

11.8 この章のまとめ

この章では、パイプを使ってプロセス間でデータをやりとりする方法について説明しました。

まず、`popen`や`pipe`を使って名前のないパイプを作成する方法を示し、パイプと`dup`を使ってあるプログラムから別のプログラムの標準入力にデータを渡す方法を説明しました。

次に、名前付きパイプを取り上げ、互いに独立したプログラムの間でデータの受け渡しを行う方法を示しました。

最後にFIFOを使ったクライアントサーバーアプリケーションを作成し、プロセスの同期化だけにとどまらず、双方向のデータフローが可能なことを示しました。

第12章

セマフォ、メッセージキュー、共有メモリ

この章では、AT&T System V.2リリースのUNIXで導入された一連のプロセス間通信機能について取り上げます。これらの機能はいずれも同じリリースで登場し、同じようなプログラミングインタフェースを備えていることから、しばしばひとまとめにして**IPC**（InterProcess Communication）機能またはSystem V IPCと呼ばれます。すでに見たように、プロセス間で通信する方法はほかにもありますが、一般にはこの章で取り上げる機能を指すのにSystem V IPC（System Vのプロセス間通信）という呼び方が使われます。

12.1 セマフォ

マルチユーザーシステムやマルチプロセッシングシステム、あるいはその両方を組み合わせた環境で動作するプログラムを記述する場合、そのコードにはしばしば、あるリソースに対して1つの実行プロセスだけが排他アクセスすることを保証しなければならない部分、いわゆる**クリティカルセクション**が存在します。

たとえば、第7章で作成した`dbm`を使ってデータベースにアクセスするアプリケーションでは、複数のプログラムが同時にデータベースの更新を試みると、データが破壊される可能性があります。しかし、2つのプログラムがそれぞれ異なるユーザーに対してデータベース用データの入力を求めても問題は起きません。つまり、コードのうち問題があるのはデータベースを更新する部分だけで、この部分がクリティカルセクションになります。

1つのリソースに複数のプログラムがアクセスする場合に起きる問題を回避するには、ある時点におけるクリティカルセクションへのアクセス権を1つの実行スレッドだけに与えるようなトークンを生成、利用する手段が必要です。

特別なハードウェアの助けを借りずに、こうした機能を実現するコードを記述することはきわめて困難です。ソフトウェアによる解決方法としては、Dekkerのアルゴリズムと呼ばれる方法があります。残念ながらこのアルゴリズムでは、ビジーウェイトあるいはスピンロックが前提となっており、メモリ上のある場所の値が変化するのを待ちながらプロセスが絶え間なく実行されます。UNIXのようなマルチタスク環境では、これはCPUリソースの無駄使いになってしまいます。

ひとつの解決策として、`open`関数に`O_EXCL`フラグを指定してファイルを作成する方法があります。この方法を使えばアトミックなファイル作成が可能になり、1つのプロセスだけがトークン（新しく作成されたファイル）を取得できます。ただし、この方法は簡単なプログラムでは十分有効ですが、複雑で高度なプログラムの場合には非常に非効率的です。

並列プログラミングにおけるこうした問題は、Dijkstraが発表した**セマフォ**という概念により、解決へ向けて大きく前進することになりました。セマフォとは、正の整数を値に取り、この値に応じてwaitとsignalの2つの操作だけが可能な特別な変数のことです。waitとsignalは、UNIXプログラミングではすでに特別な意味を持っているので、本書ではもとの表記を使って次のように表すことにします（`sv`は、セマフォ変数`semaphore variable`の略です）。

- wait はP(sv)
- signal はV(sv)

PとVはそれぞれwaitとsignalを意味するオランダ語から取ったもので、Pは*passeren*（クリティカルセクションの前のチェックポイントを“通過”）、Vは*vrijgeven*（クリティカルセクションの制御を“明け渡す”）の略です。セマフォについては、このほかに手旗信号の“上げ”と“下げ”を意味するUPとDOWNを使って説明されることもあります。

12.1.1 セマフォの定義

最も単純なセマフォは0または1の値を取る変数、いわゆる**バイナリセマフォ**で、セマフォの中では最もよく使われるものです。複数の正の整数を取るセマフォは**汎用セマフォ**と呼ばれます。この章では、主にバイナリセマフォについて説明します。

PとVの定義は非常に簡単です。セマフォ変数svがあるとすると、2つの操作はそれぞれ次のように定義できます。

- P(sv)　svが0より大きければsvをデクリメントする。svが0なら、プロセスの実行を一時停止する。
- V(sv)　svの値が0より大きくなるのを待っているプロセスがほかにあれば、そのプロセスの実行を再開する。待っているプロセスがなければ、svをインクリメントする。

別の方向から説明しましょう。セマフォ変数svは、クリティカルセクションが利用可能な場合には真です。この場合、P(sv)によってsvはデクリメントされ（svは偽になる）、クリティカルセクションはビジー状態になります。クリティカルセクションが再び利用可能になると、V(sv)によってsvがインクリメントされます。この操作を実現するには、通常の変数を用意し、これをデクリメントまたはインクリメントするだけでは不十分です。なぜなら、CやC++では、変数が真かどうかを調べて真の場合には偽を設定するという処理を1つのアトミックな操作として実現することができないからです。これは、セマフォ操作が特別な操作であることの理由でもあります。

12.1.2 セマフォのしくみ

簡単な例を示しましょう。2つのプロセスproc1とproc2があり、いずれも実行中のある時点でデータベースへの排他アクセスが必要になるとします。まず、2つのプロセスのどちらからもアクセスできるバイナリセマフォsvを定義します。svの値は最初は1です。2つのプロセスは、

クリティカルセクションにアクセスするときに同じ手順を踏む必要があります。実際には、2つのプロセスが別のプログラムである必要はなく、同じプログラムの異なるインスタンスでもかまいません。

2つのプロセスはセマフォ変数`sv`を共有します。一方のプロセスが`P(sv)`を実行すると、そのプロセスはセマフォを取得してクリティカルセクションに入ることができます。このとき、もう一方のプロセスはクリティカルセクションに入ることはできません。なぜなら、`P(sv)`を実行しようとした際、セマフォを取得することができないからです。このプロセスは、最初のプロセスがクリティカルセクションを出て`V(sv)`を実行し、セマフォを解放するまでの間待たされます。

これをコードで表すと、次のようになります。

```
semaphore sv = 1;

loop forever {
    P(sv);
    critical code section;
    V(sv);
    non-critical code section;
}
```

コード自体は非常にシンプルです。これは、`P`操作と`V`操作が十分に強力なためです。クリティカルセクションを利用するための門番として機能する`P`と`V`の操作を図にしたものを次に示します。

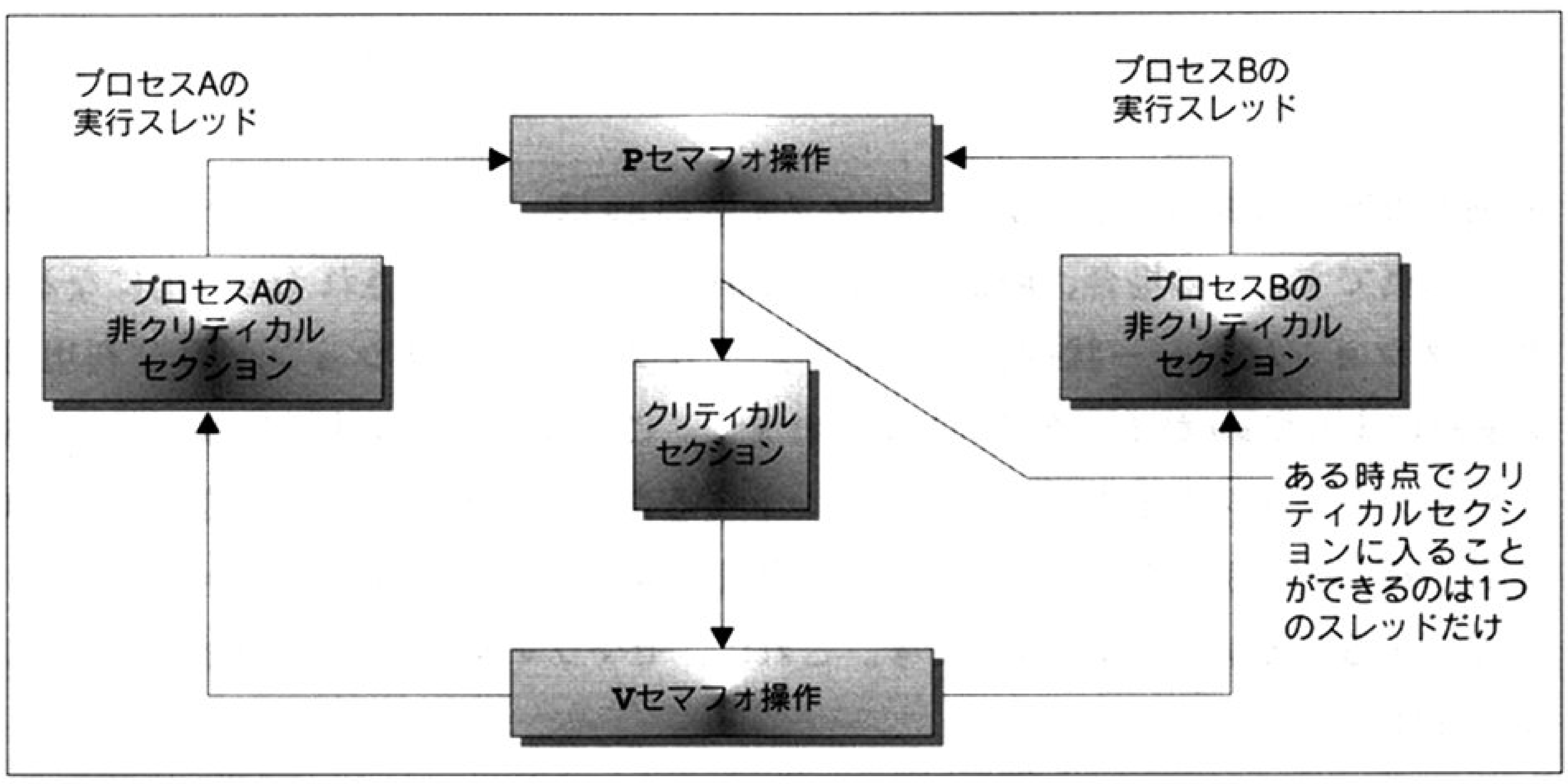


図12.1 セマフォのしくみ

12.1.3 UNIXのセマフォ機能

セマフォの概念と動作のしくみについては、上に説明したとおりです。ここでは、UNIXがセマフォ機能をどのように実装しているかを具体的に見ていくことにします。UNIXのセマフォ関数はいずれも、単一のバイナリセマフォではなく汎用セマフォの配列を操作の対象としています。このため、インタフェースはかなり込み入っており、一般に必要とされる以上の機能が提供されています。

次に示すのはセマフォ関数の定義です。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl(int sem_id, int sem_num, int command, ...);
int semget(key_t key, int num_sems, int sem_flags);
int semop(int sem_id, struct sembuf *sem_ops, size_t num_sem_ops);
```

注意

以下では各関数について説明しますが、これらの関数はセマフォ値の配列を扱うことを前提に設計されており、単一のセマフォを扱う場合とくらべてそれぞれの操作はかなり複雑になっています。以下の説明も、この点をふまえて読んでください。

`key`は、プログラムで使われるリソースを表す点でファイル名に似たものと考えることができます。プログラム間で共通の名前を使えば複数のプログラムが協調してリソースを使用できる点も、ファイル名と共通しています。同様に、`semget`が返す識別子（この識別子はほかのセマフォ関数で使われます）は、`fopen`が返すファイルストリーム`FILE *`によく似ています（ファイルストリームは、プロセスが共有ファイルにアクセスするときに使う値を意味します）。セマフォ識別子によって参照されるセマフォ自体は同じでも、異なるプロセスでは別のセマフォ識別子が使われます。この点も、ファイルの場合とよく似ています。こうしたキーと識別子の使い方は、この章で取り上げるIPC機能すべてに共通しています（ただし、共通しているのはキーと識別子の使い方であり、各機能はそれぞれ独立したキーと識別子を使います）。

◆ semget

新しいセマフォを作成したり、既存のセマフォのセマフォ識別子を取得したりするには、`semget`関数を使います。

```
int semget(key_t key, int num_sems, int sem_flags);
```

最初のパラメータkeyは、互いに関連のないプロセスが同じセマフォにアクセスするために使う整数値です。すべてのセマフォは、キーを提供するプログラムによって間接的にアクセスされますが、システムはこのキーに対してセマフォ識別子を生成します。セマフォキーを使うのはsemget関数だけです。ほかのセマフォ関数は、semgetが返すセマフォ識別子を使います。

セマフォキーには、特別な値IPC_PRIVATEがあります。keyにこの値を指定すると、セマフォを作成するプロセスだけがアクセス可能なセマフォが作成されます。この場合、セマフォを作成したプロセスが、セマフォを必要とするほかのプロセスにセマフォ識別子を直接渡す必要があります（それには、セマフォを作成したプロセスが子プロセスを作成するという方法が一般的です)。ただし、IPC_PRIVATEを指定する方法はほとんど使われません。semgetを使うときには、IPC_PRIVATEとして宣言されている値を誤ってkeyに指定しないようにチェックする必要があります。Linuxシステムでは、IPC_PRIVATEの値は0です。

パラメータnum_semsには、必要なセマフォの数を指定します。この値はほとんど常に1です。

パラメータsem_flagsには、open関数のフラグに似たフラグの集合を指定します。下位9ビットはセマフォのパーミッションで、ちょうどファイルのパーミッションと同じような意味を持ちます。フラグに値IPC_CREATとのビット単位の論理和を指定すると、新しいセマフォを作成することができます。なお、IPC_CREATフラグをセットして既存のセマフォのキーを指定してもエラーにはならず、IPC_CREATフラグが必要ない場合には単に無視されます。このため、新しい一意のセマフォを確実に取得したい場合には、IPC_CREATとIPC_EXCLの両方を指定します。こうすると、すでにセマフォが存在する場合にはエラーが返されます。

semget関数は、成功すると（0でない）正の値を返します。この値は、ほかのセマフォ関数でセマフォ識別子として使います。エラーが発生した場合には-1を返します。

◆ semop

semop関数は、セマフォの値を変更するときに使います。

```
int semop(int sem_id, struct sembuf *sem_ops, size_t num_sem_ops);
```

最初のパラメータsem_idには、semgetから返されたセマフォ識別子を指定します。2番目のパラメータsem_opsには、（少なくとも）次のようなメンバを持つ構造体の配列へのポインタを指定します。

```
struct sembuf {
    short sem_num;
    short sem_op;
    short sem_flg;
}
```

最初のメンバsem_numはセマフォ数です。セマフォの配列を操作する場合は別ですが、通常、この値は0です。メンバsem_opは、セマフォをどれだけ変更するかを表す値です。セマフォの

増減の単位は1でなくてもかまいません。通常使われるのは-1か+1のいずれかです。-1の場合は、セマフォが利用可能になるまで待つP操作に対応し、+1の場合は、セマフォが利用可能になったことを知らせるV操作に対応します。

最後のメンバ**sem_flg**には、一般に**SEM_UNDO**をセットします。**SEM_UNDO**を指定すると、現在のプロセスがセマフォに加えた変更がオペレーティングシステムによって追跡され、セマフォを解放せずにプロセスが終了した場合には、そのプロセスが保持していたセマフォがオペレーティングシステムによって自動的に解放されます。特に必要がないかぎり、**sem_flg**には**SEM_UNDO**を指定します。

semopによって要求されたすべての操作は、複数のセマフォを扱う場合に発生する可能性のあるレースコンディションを回避するために一括して処理されます。**semop**の処理についてさらに詳しく知りたい場合には、マニュアルページを参照してください。

◆ semctl

semctl関数を使うと、セマフォ情報を直接制御することができます。

```
int semctl(int sem_id, int sem_num, int command, ...);
```

最初のパラメータ**sem_id**には、**semget**から返されたセマフォ識別子を指定します。2番目のパラメータ**sem_num**には、セマフォ数を指定します。セマフォ数は、セマフォの配列を扱うときに使います。通常、この値には、最初のセマフォ1つだけを意味する0を指定します。パラメータ**command**には、実行する操作を指定します。4番目のパラメータが存在する場合、このパラメータは**union semun**です。共用体**semun**は、少なくとも次のようなメンバを持ちます。

```
union semun {
    int val;
    struct semid_ds *buf;
    unsigned short *array;
}
```


semctlで指定できる**command**の値にはさまざまなものがあります。ここでは、そのうちよく使われる2つの値だけを取り上げます。その他の値や**semctl**の動作の詳細については、マニュアルページを参照してください。

commandの値としてよく使うのは、次の2つです。

- **SETVAL**　セマフォを既知の値に初期化する。値は、共用体**semun**のメンバ**val**で指定する。セマフォを機能させるには、セマフォを初めて使う前にこの操作を行う必要がある。
- **IPC_RMID**　不要になったセマフォ識別子を削除する。

semctl関数は、パラメータcommandによって異なる値を返します。上に示したSETVALとIPC_RMIDの場合には、成功すると0を返し、失敗すると-1を返します。

12.1.4 セマフォの使い方

上の説明からわかるように、セマフォ操作はかなり面倒です。これは非常に残念なことです。というのも、クリティカルセクションを持つ複数のプロセスを対象としたプログラミング自体、すでに相当複雑な側面を持っているからです。

ただし、セマフォを必要とするような問題の多くは、セマフォの最も単純な形式であるバイナリセマフォで解決することができます。以下では、上に説明したプログラミングインタフェースを使って、バイナリセマフォに特化したP操作とV操作用のシンプルなインタフェースを作成し、これを使ってセマフォ操作の具体例を示すことにします。

サンプルプログラムは1つしか作成しませんが、繰り返し呼び出すことによってセマフォの動作を確かめます。プログラムには引数を指定することができます。引数を指定して呼び出されたプログラムでは、セマフォの作成と削除を行います。

プログラムでは、クリティカルセクションへの出入りを表すために2つの異なる文字を使います。オプション引数を指定して起動した場合には、Xを使ってクリティカルセクションへの出入りを表し、オプションを指定せずに起動した場合にはOを使います。クリティカルセクションに入ることができるのは1つのプロセスだけなので、XとOは常にペアで表示されるはずです。作成するプログラムの名前はsem1.cとします。

例題 セマフォ

1 ヘッダーファイルをインクルードし、プロトタイプ宣言を行ってグローバル変数を記述します。main関数では、semgetを呼び出してセマフォを作成します。最初に呼び出すプログラムでは引数を指定します。この場合には、argc > 1が真になるので、set_semvalueが呼び出され、セマフォが初期化されます。また、op_charにXが設定されます。

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

static int set_semvalue(void);
static void del_semvalue(void);
static int semaphore_p(void);
static int semaphore_v(void);
static int sem_id;
```

```
int main(int argc, char *argv[])
{
    int i;
    int pause_time;
    char op_char = 'O';

    srand((unsigned int)getpid());

    sem_id = semget((key_t)1234, 1, 0666 | IPC_CREAT);

    if (argc > 1) {
        if (!set_semvalue()) {
            fprintf(stderr, "Failed to initialize semaphore\n");
            exit(EXIT_FAILURE);
        }
        op_char = 'X';
        sleep(2);
    }
```

2 ループを用意し、クリティカルセクションへの出入りを10回繰り返します。ループの中では、クリティカルセクションに入るときにsemaphore_pを呼び出します。

```
    for(i = 0; i < 10; i++) {
        if (!semaphore_p()) exit(EXIT_FAILURE);
        printf("%c", op_char);fflush(stdout);
        pause_time = rand() % 3;
        sleep(pause_time);
        printf("%c", op_char);fflush(stdout);
```

3 クリティカルセクションを抜けたら、semaphore_vを呼び出してセマフォを利用可能にします。次のループを開始する前に、ランダムな時間待機します。ループを終えたら、del_semvalueを呼び出して不要になったセマフォ識別子を削除します。

```
        if (!semaphore_v()) exit(EXIT_FAILURE);
        pause_time = rand() % 2;
        sleep(pause_time);
    }
    printf("\n%d - finished\n", getpid());
    if (argc > 1) {
        sleep(10);
        del_semvalue();
    }
    exit(EXIT_SUCCESS);
}
```

4 set_semvalue関数では、SETVALコマンドを指定してsemctlを呼び出し、セマフォを初期化します。この操作はセマフォを使う前に行う必要があります。

```
static int set_semvalue(void)
{
    union semun sem_union;

    sem_union.val = 1;
    if (semctl(sem_id, 0, SETVAL, sem_union) == -1) return(0);
    return(1);
}
```

5 del_semvalue関数もset_semvalue関数とほとんど同じですが、IPC_RMIDコマンドを指定してsemctlを呼び出し、セマフォ識別子を削除します。

```
static void del_semvalue(void)
{
    union semun sem_union;

    if (semctl(sem_id, 0, IPC_RMID, sem_union) == -1)
        fprintf(stderr, "Failed to delete semaphore\n");
}
```

6 semaphore_p関数はセマフォの値を1つ減らします (-1)。

```
static int semaphore_p(void)
{
    struct sembuf sem_b;

    sem_b.sem_num = 0;
    sem_b.sem_op = -1; /* P() */
    sem_b.sem_flg = SEM_UNDO;
    if (semop(sem_id, &sem_b, 1) == -1) {
        fprintf(stderr, "semaphore_p failed\n");
        return(0);
    }
    return(1);
}
```

7 semaphore_v関数では、sembuf構造体のメンバsem_opに1を設定し、セマフォが利用可能になるようにします。

```
static int semaphore_v(void)
{
    struct sembuf sem_b;

    sem_b.sem_num = 0;
    sem_b.sem_op = 1; /* V() */
    sem_b.sem_flg = SEM_UNDO;
    if (semop(sem_id, &sem_b, 1) == -1) {
        fprintf(stderr, "semaphore_v failed\n");
        return(0);
```

```
    }
    return(1);
}
```

このプログラムでは、一度に1つのバイナリセマフォしか使用できません。プログラムを拡張してセマフォ変数を渡すようにすれば、もっと多くのセマフォを扱うこともできますが、ほとんどの場合には1つのバイナリセマフォだけで十分です。

セマフォの動作を確認するために、プログラムを複数回呼び出します。最初に呼び出すときには引数を指定し、呼び出されたプログラムがセマフォの作成と削除を行うようにします。以後に呼び出すプログラムでは引数を指定しません。

次に示すのは、プログラムを2回続けて呼び出したときの出力例です。

```
$ sem1 1 &
[1] 1082
$ sem1
OOXXOOXXOOXXOOXXOOXXOOOOXXOOXXOOXXOOXXXX
1083 - finished
1082 - finished
$
```

oと**x**がペアになって出力されており、クリティカルセクションが正しく処理されていることがわかります。

解説

このプログラムでは、まず**semget**関数を使って、適当に選んだキーからセマフォ識別子を取得します。**IPC_CREAT**フラグを指定しているので、必要に応じてセマフォが作成されます。

プログラムを呼び出すときに引数が指定されていた場合には、**set_semvalue**関数でセマフォを初期化します。**set_semvalue**は、汎用的な**semctl**関数に対するよりシンプルなインタフェースとして用意した関数です。また、プログラムからの出力に使う文字も引数の有無によって決定します。続けて起動されるプログラムを待つために、**sleep**を挿入しています。**srand**と**rand**を使っているのは、プログラムの実行タイミングをランダムにするためです。

プログラムでは、クリティカルセクションと非クリティカルセクションの両方でランダムな時間待機しながら、ループを10回実行します。クリティカルセクションの前後で呼び出している**semaphor_p**と**semaphore_v**は、**semop**関数に対するシンプルなインタフェースとして用意した関数です。

パラメータを指定して呼び出されたプログラムでは、セマフォを削除する前にしばらく待機し、ほかのプログラムが終了するを待ちます。セマフォが削除されなかった場合、そのセマフォは（たとえプログラムで使われていなくても）システムに残ります。したがって、セマフォを使うプログラムを作成するときには、プログラムの終了後にセマフォが（意図に反して）残らないように注意する必要があります。もしセマフォが残っていると、次にプログラムを実行したときに問

題が起きる可能性があります。また、セマフォ自体限られたリソースなので、セマフォが残っているとリソースを無駄使いすることになります。

12.1.5 セマフォのまとめ

セマフォのプログラミングインタフェースは複雑ですが、本書ではこれを単純化したインタフェースを例として示しました。セマフォを使うほとんどのプログラムでは、このインタフェースで十分に目的を達することができます。

12.2 共有メモリ

共有メモリもIPC機能のひとつです。共有メモリを使うと、関連のない複数のプロセスが同じ論理メモリにアクセスできるようになります。共有メモリは、実行中の2つのプロセスの間でデータを転送するための非常に効率的な手段です。共有メモリの実装のうちほとんどのものは、2つのプロセス間で共有されるメモリに対して同じ物理メモリを使うようです（ただし、X/Open仕様ではこれは必要とされていません）。

12.2.1 概要

共有メモリは、あるプロセスのためにIPCによって作成される特別な範囲のアドレスで、このアドレス範囲はそのプロセスのアドレス空間にあります。ほかのプロセスは、同じ共有メモリセグメントを自分のアドレス空間に“アタッチ”することができます。共有メモリセグメントをアタッチしたすべてのプロセスは、`malloc`で割り当てられたメモリと同じように共有メモリにアクセスできます。あるプロセスが共有メモリに対して書き込みを行った場合、その変更内容は、同じ共有メモリにアクセスするほかのすべてのプロセスからただちに見えるようになります。

共有メモリ自体は同期機能を提供していません。つまり、最初のプロセスによる書き込みが終わらないうちに別のプロセスが共有メモリを読み取ることを防ぐ機能は用意されていません。複数のプロセスからのアクセスの同期は、プログラマが行わなければなりません。

次の図の矢印は、各プロセスの論理アドレス空間から物理アドレス空間へのマッピングを表しています。利用可能メモリは物理メモリとディスクにスワップアウトされたメモリページから構成されるので、実際のマッピングは図よりもはるかに複雑になります。

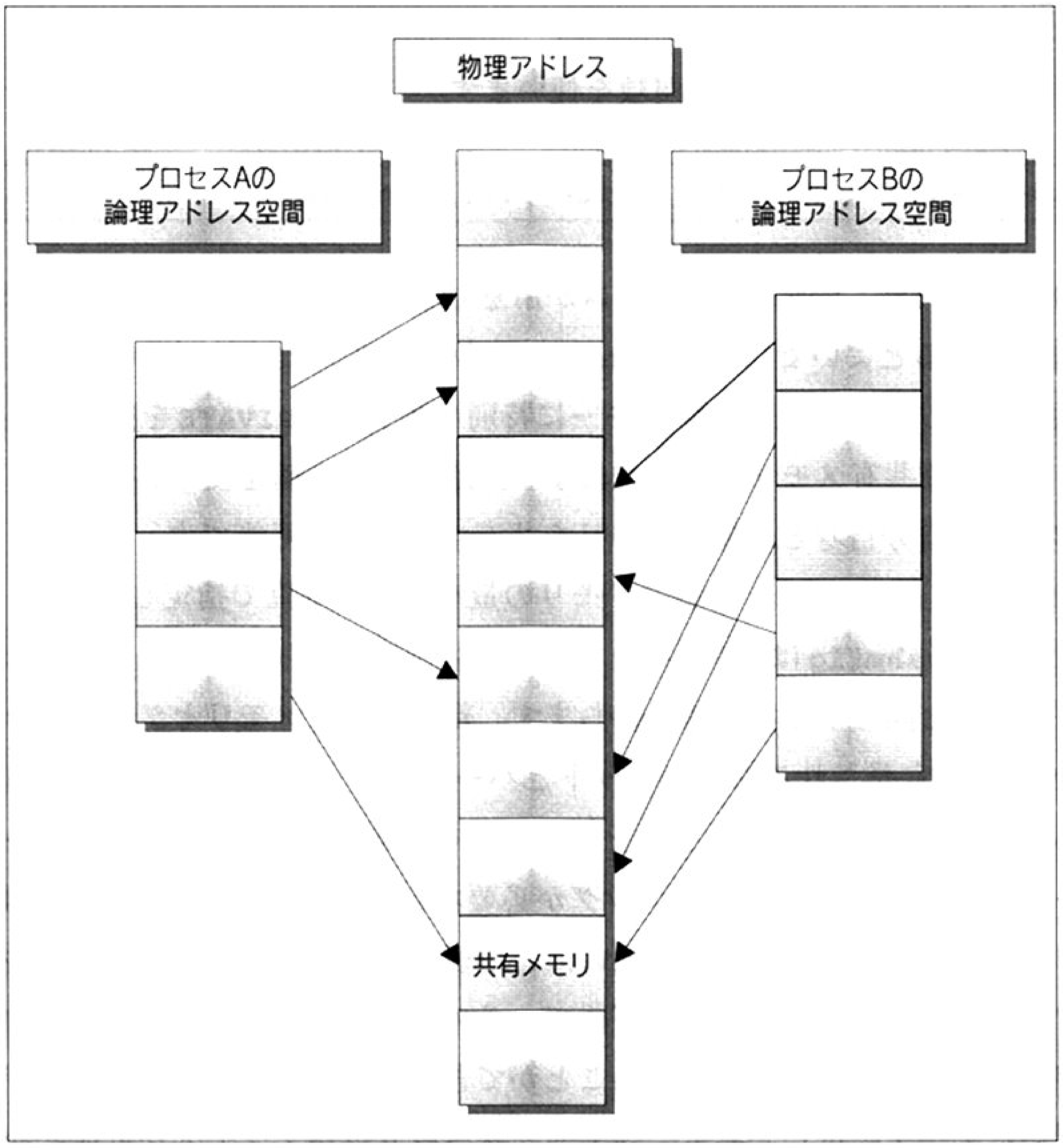


図12.2 共有メモリ

12.2.2 共有メモリ関数

共有メモリ関連関数は、セマフォ操作関数とよく似ています。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

void *shmat(int shm_id, const void *shm_addr, int shmflg);
int shmctl(int shm_id, int cmd, struct shmid_ds *buf);
int shmdt(const void *shm_addr);
int shmget(key_t key, size_t size, int shmflg);
```

以下ではそれぞれの関数について説明します。

◆ shmget

共有メモリを作成するにはshmget関数を使います。

```
int shmget(key_t key, size_t size, int shmflg);
```

セマフォの場合と同様、keyはプログラムが用意します。このキーは、共有メモリセグメントに付ける名前と考えるとよいでしょう。shmget関数は共有メモリ識別子を返します。この識別子はほかの共有メモリ関数で使います。キーに特別な値IPC_PRIVATEを指定すると、そのプロセスにプライベートな共有メモリが作成されます。通常、この値を使うことはありませんが、ある単一のプロセスが自分自身との間でメモリを共有することは可能です。

2番目のパラメータsizeには、必要なメモリの量をバイト単位で指定します。

3番目のパラメータshmflgは9つのパーミッションフラグから構成され、ファイルを作成するときのモードフラグと同じような意味を持ちます。新しい共有メモリセグメントを作成するときには、IPC_CREATで定義される特別なビットとパーミッションとのビット単位の論理和を指定する必要があります。IPC_CREATフラグをセットして既存の共有メモリセグメントのキーを指定してもエラーにはならず、IPC_CREATフラグが必要ない場合には単に無視されます。

共有メモリのパーミッションフラグは大いに利用価値があります。たとえば、共有メモリの作成者が所有するプロセスからは書き込み可能で、その他のユーザーが作成したプロセスからは読み取りだけが可能な共有メモリを作成することができます。パーミッションフラグを使えば、共有メモリに置いたデータに対して非常に効率的な読み取り専用アクセスが可能になります。もちろん、ほかのユーザーによってデータが変更される心配もありません。

共有メモリの作成に成功すると、shmgetは負ではない整数（共有メモリ識別子）を返します。失敗した場合には-1を返します。

◆ shmat

共有メモリセグメントを作成しただけでは、どのプロセスからもその共有メモリにアクセスすることはできません。共有メモリにアクセスできるようにするには、プロセスのアドレス空間に共有メモリをアタッチする必要があります。これを行うのが次のshmat関数です。

```
void *shmat(int shm_id, const void *shm_addr, int shmflg);
```

最初のパラメータshm_idには、shmgetから返された共有メモリ識別子を指定します。

2番目のパラメータshm_addrには、現在のプロセスに共有メモリをアタッチするときのアドレスを指定します。通常はヌルポインタを指定し、アドレスの選択をシステムに任せます。

3番目のパラメータshmflgにはビット単位のフラグを指定します。指定できる値は、SHM_RNDとSHM_RDONLYの2つです。SHM_RNDはshm_addrと組み合わせて、共有メモリをアタッチするアドレスを制御するのに使います。SHM_RDONLYは、アタッチされた共有メモリを読み取り専用にします。共有メモリをアタッチするアドレスの指定が必要になるケースはごくまれであり、通

常はアドレスの選択をシステムに任せます（アドレスを指定すると、アプリケーションのハードウェア依存度がきわめて高くなってしまいます）。

shmatは、成功すると共有メモリの先頭バイトへのポインタを返します。失敗した場合には-1を返します。

共有メモリに対する読み取りアクセスまたは書き込みアクセスが可能かどうかは、オーナー（共有メモリの作成者）、パーミッション、および現在のプロセスのオーナーによって決まります。共有メモリに対するパーミッションは、ファイルに対するパーミッションと多くの類似点を持っています。

ただし、**shmflg & SHM_RDONLY**が真の場合は例外で、パーミッションが書き込みアクセスを許可する設定であっても共有メモリに書き込むことはできません。

◆ shmdt

shmdt関数は、現在のプロセスから共有メモリを"デタッチ"します。引数には、**shmat**から返されたアドレスを指定します。**shmdt**は成功すると0を返し、失敗した場合には-1を返します。共有メモリをデタッチしても、現在のプロセスから共有メモリを利用できなくなるだけで、共有メモリが削除されるわけではありません。

◆ shmctl

共有メモリを制御するための関数は、セマフォを制御する関数よりも単純です。

```
int shmctl(int shm_id, int command, struct shmid_ds *buf);
```

shmid_ds構造体には、少なくとも次のメンバがあります。

```
struct shmid_ds {
    uid_t shm_perm.uid;
    uid_t shm_perm.gid;
    mode_t shm_perm.mode;
}
```

最初のパラメータ**shm_id**には、**shmget**から返された共有メモリ識別子を指定します。

2番目のパラメータ**command**には、実行する操作を指定します。次の3つの値を指定できます。

表12.1 shmid_ds構造体のcommandパラメータ

コマンド	説明
IPC_STAT	共有メモリに関連付けられている値を取得し、**shmid_ds**構造体に格納する。
IPC_SET	**shmid_ds**構造体で指定された値に基づいて、共有メモリに関連付けられている値を設定する。 プロセスは、この操作を行うためのパーミッションを持っている必要がある。
IPC_RMID	共有メモリセグメントを削除する。

3番目のパラメータbufには、共有メモリのモードとパーミッションを含む構造体へのポインタを指定します。

shmctl関数は成功すると0を返し、失敗した場合には-1を返します。X/Open仕様では、アタッチされている共有メモリの削除を試みた場合の動作については記述がありません。一般に、アタッチされている共有メモリを削除した場合、その共有メモリは最後のプロセスからデタッチされるまで機能し続けます。ただし、これは記述のない動作なので、このような動作を前提にプログラムを作成することは避けるべきです。

では、共有メモリ関数を使ってサンプルプログラムを作成してみましょう。ここでは、shm1.cとshm2.cの2つのプログラムを作成します。shm1.cは消費者で、共有メモリセグメントを作成し、共有メモリに書き込まれたデータを表示します。shm2.cは生産者で、既存の共有メモリセグメントをアタッチし、このセグメントにデータを入力します。

例題 共有メモリ

1 最初に示すのは、生産者となるshm1.cです。必要なヘッダーファイルをインクルードし、適切なMEM_SZと構造体を定義します。次に、IPC_CREATを指定してshmgetを呼び出し、MEM_SZで指定されたサイズの共有メモリセグメントを作成します。

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define MEM_SZ 4096

struct shared_use_st {
    int written_by_you;
    char some_text[BUFSIZ];
};

int main()
{
    int running = 1;
    void *shared_memory = (void *)0;
    struct shared_use_st *shared_stuff;
    int shmid;

    srand((unsigned int)getpid());

    shmid = shmget((key_t)1234, MEM_SZ, 0666 | IPC_CREAT);

    if (shmid == -1) {
        fprintf(stderr, "shmget failed\n");
```

```
        exit(EXIT_FAILURE);
    }
```

2 共有メモリをプログラムからアクセスできるようにします。

```
    shared_memory = shmat(shmid, (void *)0, 0);
    if (shared_memory == (void *)-1) {
        fprintf(stderr, "shmat failed\n");
        exit(EXIT_FAILURE);
    }
    printf("Memory attached at %X\n", (int)shared_memory);
```

3 shared_memoryをshared_stuffに代入し、some_textにテキストがあれば、そのテキストを表示します。some_textの内容がendだった場合にはループを抜けます。sleepを呼び出しているので、消費者は一定時間クリティカルセクションにいすわります。この間、生産者は待たされることになります。

```
    shared_stuff = (struct shared_use_st *)shared_memory;
    shared_stuff->written_by_you = 0;
    while(running) {
        if (shared_stuff->written_by_you) {
            printf("You wrote: %s", shared_stuff->some_text);
            sleep( rand() % 4 );
            shared_stuff->written_by_you = 0;
            if (strncmp(shared_stuff->some_text, "end", 3) == 0) {
                running = 0;
            }
        }
    }
```

4 最後に、共有メモリをデタッチして削除します。

```
    if (shmdt(shared_memory) == -1) {
        fprintf(stderr, "shmdt failed\n");
        exit(EXIT_FAILURE);
    }
    if (shmctl(shmid, IPC_RMID, 0) == -1) {
        fprintf(stderr, "shmctl(IPC_RMID) failed\n");
        exit(EXIT_FAILURE);
    }
    exit(EXIT_SUCCESS);
}
```

5 次のshm2.cは生産者で、消費者のためにデータを入力します。コードの大部分はshm1.cと同じです。

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define MEM_SZ 4096

struct shared_use_st {
    int written_by_you;
    char some_text[BUFSIZ];
};

int main()
{
    int running = 1;
    void *shared_memory = (void *)0;
    struct shared_use_st *shared_stuff;
    char buffer[BUFSIZ];
    int shmid;

    shmid = shmget((key_t)1234, MEM_SZ, 0666 | IPC_CREAT);

    if (shmid == -1) {
        fprintf(stderr, "shmget failed\n");
        exit(EXIT_FAILURE);
    }
    shared_memory = shmat(shmid, (void *)0, 0);
    if (shared_memory == (void *)-1) {
        fprintf(stderr, "shmat failed\n");
        exit(EXIT_FAILURE);
    }
    printf("Memory attached at %X\n", (int)shared_memory);

    shared_stuff = (struct shared_use_st *)shared_memory;
    while(running) {
        while(shared_stuff->written_by_you == 1) {
            sleep(1);
            printf("waiting for client...\n");
        }
        printf("Enter some text: ");
        fgets(buffer, BUFSIZ, stdin);

        strcpy(shared_stuff->some_text, buffer);
        shared_stuff->written_by_you = 1;

        if (strncmp(buffer, "end", 3) == 0) {
                running = 0;
        }
    }
    if (shmdt(shared_memory) == -1) {
```

```
        fprintf(stderr, "shmdt failed\n");
        exit(EXIT_FAILURE);
    }
    exit(EXIT_SUCCESS);
}
```

出力例を次に示します。

```
$ shm1 &
[1] 294
Memory attached at 50007000
$ shm2
Memory attached at 50007000
Enter some text: hello
You wrote: hello
waiting for client...
waiting for client...
Enter some text: Linux!
You wrote: Linux!
waiting for client...
waiting for client...
waiting for client...
Enter some text: end
You wrote: end
$
```

解説

最初に起動するプログラム**shm1**は、共有メモリセグメントを作成し、これを自分のアドレス空間にアタッチします。次に、共有メモリの最初の部分に**shared_use_st**構造体を重ね合わせます。この構造体には**written_by_you**というフラグがあり、データが利用可能になると、このフラグがセットされます。フラグがセットされている場合、プログラムはテキストを読み取って表示します。その後、データの読み取りが終わったことを示すためにフラグをクリアします。文字列**end**は、ループから抜けるための合図として使います。ループを抜けたら、共有メモリセグメントをデタッチして削除します。

次に起動するプログラム**shm2**は、同じキー**1234**を使用し、**shm1**と同じ共有メモリセグメントを取得してアタッチします。次に、ユーザーにテキストの入力を求めます。フラグ**written_by_you**がセットされている場合には、クライアントプロセスが前のデータをまだ読み取っていないので、データが読み取られるまで待機します。クライアントプロセスによってフラグがクリアされたら、**shm2**は新しいデータを書き込んでフラグをセットします。文字列**end**が入力されたら、ループを抜けて共有メモリをデタッチします。

このサンプルプログラムでは、独自の同期化フラグとして**written_by_you**を用意しました。しかし、絶えずループを回るために非常に非効率的なビジーウェイト状態が発生してしまい、同期化フラグとしてはあまり望ましくありません。実用的なプログラムを作成する場合には、メッ

12

セージを渡す（パイプ、または次に説明するIPCメッセージを使用）、シグナルを生成する、セマフォを使って読み手側と書き手側を同期させるなどの方法を検討する必要があります。

12.2.3 共有メモリのまとめ

共有メモリは、複数のプロセス間でデータの共有や受け渡しを行うための効率的な手段です。ただし、同期機能は提供されないので、一般にはほかのなんらかのメカニズムを使って共有メモリに対するアクセスを同期化する必要があります。共有メモリは、広いメモリ領域に効率的にアクセスする目的で使われることが多く、共有メモリへのアクセスの同期化は、短いメッセージをやりとりして実現するのが一般的です。

12.3 メッセージキュー

IPC機能には、これまでに説明したセマフォと共有メモリに加えて、**メッセージキュー**というものがあります。

12.3.1 概要

メッセージキューは多くの点で名前付きパイプに似ていますが、パイプのオープンやクローズといった面倒な手順は必要ありません。ただし、メッセージを使えば名前付きパイプの問題がすべて解決されるというわけではありません。

メッセージキューを使うと、あるプロセスから別のプロセスにデータのブロックを送信することができます。各データブロックは型を持つものとして扱われ、受信側のプロセスではさまざまな型を持つデータのブロックを受信することができます。メッセージキューの利点は、名前付きパイプの持つ同期化の問題やブロックに関する問題をほぼ完全に回避できる点です。逆に欠点は、パイプの場合と同様、各データブロックに最大サイズに関する制限があり、またシステム全体を通じてすべてのキュー上のすべてのブロックの最大合計サイズにも制限があるという点です。

メッセージキューにはこうした制限がありますが、X/Open仕様では、制限を超えると一部のメッセージキュー関数が失敗すると規定されているだけで、制限自体がどのようなものなのかを知る方法については記述されていません。Linuxでは、`MSGMAX`（4056）と`MSGMNB`（16384）が定義されており、それぞれ各メッセージのバイト単位の最大サイズ、1つのキューの最大サイズを表します。これらの値はシステムによって異なる可能性があり、またシステムによっては定義自体が存在しないこともあります。

12.3.2 メッセージキュー関数

メッセージキュー関数の定義は次のとおりです。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl(int msqid, int cmd, struct msqid_ds *buf);
int msgget(key_t key, int msgflg);
int msgrcv(int msqid, void *msg_ptr, size_t msg_sz, long int msgtype, int msgflg);
int msgsnd(int msqid, const void *msg_ptr, size_t msg_sz, int msgflg);
```

◆ msgget

メッセージキューを作成したり、メッセージキューにアクセスしたりするには、**msgget**関数を使います。

```
int msgget(key_t key, int msgflg);
```

セマフォや共有メモリの場合と同様、メッセージキューに名前を付けるための**key**の値はプログラムが用意します。キーに特別な値**IPC_PRIVATE**を指定すると、現在のプロセスだけがアクセスできるプライベートなキューが作成されます。2番目のパラメータ**msgflg**は9つのパーミッションフラグから構成されます。新しいメッセージキューを作成するときには、**IPC_CREAT**で定義される特別なビットとパーミッションとのビット単位の論理和を指定する必要があります。**IPC_CREAT**フラグをセットして既存のメッセージキューのキーを指定してもエラーにはならず、**IPC_CREAT**フラグが必要ない場合には単に無視されます。

msggetは、成功すると正の整数(キュー識別子)を返します。失敗した場合には -1を返します。

12

◆ msgsnd

msgsnd関数は、メッセージキューにメッセージを追加します。

```
int msgsnd(int msqid, const void *msg_ptr, size_t msg_sz, int msgflg);
```

メッセージの構造は2つの点で制限を受けます。まず、メッセージはシステムの制限より小さくなければなりません。次に、必ず**long int**で始まらなければなりません。この**long int**は、メッセージ受信関数でメッセージの型として使われます。通常、メッセージを使うときには次のような構造体を定義します。

```
struct my_message {
    long int message_type;
    /* The data you wish to transfer */
}
```

message_typeはメッセージを受信するときに使われるので、このメンバを無視することはできません。メッセージのデータ構造を宣言するときはこのメンバを含めなければなりませんが、それだけでなく、既知の値を持つように初期化しておくほうが賢明です。

msgsnd関数の最初のパラメータ**msqid**には、**msgget**から返されたメッセージキュー識別子を指定します。

2番目のパラメータ**msg_ptr**には、送信するメッセージへのポインタを指定します。すでに説明したように、メッセージは**long int**型で始まらなければなりません。

3番目のパラメータ**msg_sz**には、**msg_ptr**によって示されるメッセージのサイズを指定します。メッセージのサイズには、**long int**のメッセージ型の部分は含まれません。

4番目のパラメータ**msgflg**は、現在のメッセージキューがいっぱいだったり、キューに入っているメッセージに対するシステム全体での制限に達した場合の動作を指定するフラグです。**msgflg**で**IPC_NOWAIT**フラグがセットされている場合、**msgsnd**関数はメッセージを送信せずにただちに戻ります。この場合、関数からの戻り値は -1になります。**msgflg**で**IPC_NOWAIT**フラグがクリアされている場合、送信側のプロセスはキューに空きができるまで一時停止します。

msgsnd関数は成功すると0を返し、失敗した場合には -1を返します。呼び出しが成功した場合には、メッセージデータのコピーが取り出され、メッセージキューに置かれます。

◆ msgrcv

msgrcv関数は、メッセージキューからメッセージを取得します。

```
int msgrcv(int msqid, void *msg_ptr, size_t msg_sz, long int msgtype, int msgflg);
```

最初のパラメータ**msqid**には、**msgget**から返されたメッセージキュー識別子を指定します。

2番目のパラメータ**msg_ptr**には、受信するメッセージへのポインタを指定します。**msgsnd**ですでに説明したように、メッセージは**long int**型で始まらなければなりません。

3番目のパラメータ**msg_sz**には、**msg_ptr**によって示されるメッセージのサイズを指定します。メッセージのサイズには、**long int**のメッセージ型の部分は含まれません。

4番目のパラメータ**msgtype**は**long int**型で、このパラメータを使うことで簡単な受信の優先度を指定できます。具体的には、**msgtype**が0ならキュー内で最初に利用可能なメッセージを取り出します。値が0より大きい場合には、同じメッセージ型の最初のメッセージを取り出します。値が0より小さい場合には、**msgtype**の絶対値と同じか、またはそれより小さい型の最初のメッセージを取り出します。

この説明ではわかりにくいかもしれません。実際にメッセージを受信する側から解説を加えま

しょう。まず、送信された順序でメッセージを受信したい場合には、**msgtype**に0を設定します。特定のメッセージ型のメッセージだけを受信したい場合には、該当する型の値に等しい値を**msgtype**に設定します。型が**n**か、またはそれより小さいメッセージを受信したい場合には、**msgtype**に**-n**を設定します。

5番目のパラメータ**msgflg**は、該当する型のメッセージがない場合の動作を指定するフラグです。**msgflg**で**IPC_NOWAIT**フラグがセットされている場合、**msgrcv**関数はただちに戻ります。この場合、関数からの戻り値は-1になります。**msgflg**で**IPC_NOWAIT**フラグがクリアされている場合には、該当する型のメッセージが到着するまで、プロセスは一時停止します。

操作に成功すると、**msgrcv**関数は受信バッファに置かれたバイト数を返します。このとき、**msg_ptr**によって示されるバッファにメッセージがコピーされ、メッセージキューからデータが削除されます。失敗した場合には-1を返します。

◆ msgctl

メッセージキュー関数の最後の関数**msgctl**は、共有メモリを制御する関数によく似ています。

```
int msgctl(int msqid, int command, struct msqid_ds *buf);
```

msqid_ds構造体には、少なくとも次のメンバがあります。

```
struct msqid_ds {
    uid_t  msg_perm.uid;
    uid_t  msg_perm.gid
    mode_t msg_perm.mode;
}
```

最初のパラメータ**msqid**には、**msgget**から返されたメッセージキュー識別子を指定します。

2番目のパラメータ**command**には、実行する操作を指定します。次の3つの値を指定できます。

表12.2 msgctl構造体のcommandパラメータ

コマンド	説明
IPC_STAT	メッセージキューに関連付けられている値を取得し、msqid_ds構造体に格納する。
IPC_SET	msqid_ds構造体で指定された値に基づいて、メッセージキューに関連付けられている値を設定する。プロセスは、この操作を行うためのパーミッションを持っている必要がある。
IPC_RMID	メッセージキューを削除する。

msgctl関数は成功すると0を返し、失敗した場合には-1を返します。プロセスが**msgsnd**または**msgrcv**関数で待っているときにメッセージキューが削除された場合、送信関数または受信関数は操作に失敗します。

以上でメッセージキュー関数についての説明は終わりです。さっそくサンプルプログラムを作成してメッセージキューの動作を確かめてみましょう。2つのプログラム**msg1.c**と**msg2.c**を作

成します。msg1.cはメッセージを受信するプログラム、msg2.cはメッセージを送信するプログラムです。メッセージキューの作成はどちらのプログラムでも可能ですが、メッセージキューの削除は、受信側が最後のメッセージを受信したあとに受信側のプログラムで実行します。

例題 メッセージキュー

1 最初に示すのは、受信側のプログラムmsg1.cです。

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct my_msg_st {
    long int my_msg_type;
    char some_text[BUFSIZ];
};

int main()
{
    int running = 1;
    int msgid;
    struct my_msg_st some_data;
    long int msg_to_receive = 0;
```

2 まず、メッセージキューをセットアップします。

```
    msgid = msgget((key_t)1234, 0666 | IPC_CREAT);

    if (msgid == -1) {
        fprintf(stderr, "msgget failed\n");
        exit(EXIT_FAILURE);
    }
```

3 次に、endというメッセージに出会うまで、キューからメッセージを取り出します。最後にメッセージキューを削除します。

```
    while(running) {
        if (msgrcv(msgid, (void *)&some_data, BUFSIZ,
                   msg_to_receive, 0) == -1) {
            fprintf(stderr, "msgrcv failed\n");
            exit(EXIT_FAILURE);
        }
        printf("You wrote: %s", some_data.some_text);
        if (strncmp(some_data.some_text, "end", 3) == 0) {
```

```
            running = 0;
        }
    }
    if (msgctl(msgid, IPC_RMID, 0) == -1) {
        fprintf(stderr, "msgctl(IPC_RMID) failed\n");
        exit(EXIT_FAILURE);
    }
    exit(EXIT_SUCCESS);
}
```

4 次に示すのは、送信側のプログラム**msg2.c**です。**msg1.c**とほとんど同じですが、**msg_to_receive**の代わりに**buffer[BUFSIZ]**を宣言し、メッセージキューを削除する部分を省略しています。また、ループの中も書き換え、**msgsnd**を呼び出して入力されたテキストをキューに送信します。

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct my_msg_st {
    long int my_msg_type;
    char some_text[BUFSIZ];
};

int main()
{
    int running = 1;
    struct my_msg_st some_data;
    int msgid;
    char buffer[BUFSIZ];

    msgid = msgget((key_t)1234, 0666 | IPC_CREAT);

    if (msgid == -1) {
        fprintf(stderr, "msgget failed\n");
        exit(EXIT_FAILURE);
    }
    while(running) {
        printf("Enter some text: ");
        fgets(buffer, BUFSIZ, stdin);
        some_data.my_msg_type = 1;
        strcpy(some_data.some_text, buffer);

        if (msgsnd(msgid, (void *)&some_data, BUFSIZ, 0) == -1) {
            fprintf(stderr, "msgsnd failed\n");
            exit(EXIT_FAILURE);
        }
```

12

```
        if (strncmp(buffer, "end", 3) == 0) {
            running = 0;
        }
    }
    exit(EXIT_SUCCESS);
}
```

パイプの場合とは異なり、プロセスが自分自身で同期化の手段を用意する必要はありません。これは、パイプに対するメッセージの大きな利点のひとつです。

メッセージキューに空きがある場合、送信側はキューを作成し、キューにデータを入れてから、自分自身は受信側の起動前に終了することができます。ここでは、最初に送信側の**msg2**を実行します。実行例を次に示します。

```
$ msg2
Enter some text: hello
Enter some text: How are you today?
Enter some text: end
$ msg1
You wrote: hello
You wrote: How are you today?
You wrote: end
$
```

解説

送信側プログラムは、**msgget**関数を使ってメッセージキューを作成し、**msgsnd**関数を使ってキューにメッセージを追加します。受信側プログラムは、**msgget**を使ってメッセージキュー識別子を取得し、メッセージを受信します。**end**という特別なテキストを受け取ると、メッセージの受信を終え、**msgctl**を使ってメッセージキューを削除します。

12.3.3 キューの効率

上の2つのサンプルプログラムから不要な部分を削除して最低限必要なコードだけを残し、10MBのデータをやりとりしてキューの効率を測定してみました。その結果、筆者のマシンでは処理に1.9秒ほどかかりました。パイプを使った場合よりかなり遅くなっていますが、それでも非常によい数字といえます。システムによっては、メッセージキューのほうがパイプより速い場合もあります。

12.3.4 メッセージキューのまとめ

メッセージキューは、互いに関連のない2つのプロセスの間でデータをやりとりするための効率的な方法で、使い方も比較的簡単です。パイプと比べた場合のメッセージキューの利点は、メッセージキューが送信側と受信側のどちらのプロセスからも独立して存在することです。このため、名前付きパイプでのオープンとクローズの同期化といった問題も、メッセージキューでは考慮せずにすみます。

12.4 CDデータベースアプリケーション

これまでに取り上げたIPC機能を使ってCDデータベースアプリケーションを書き直してみましょう。

IPC機能は3種類あり、さまざまな組み合わせが可能ですが、CDデータベースアプリケーションではごく少量の情報をやりとりするだけなので、メッセージキューを使って要求と応答を直接送受信することにします。

やりとりする情報が多い場合には、実際に受け渡しする情報を共有メモリに置き、必要なデータが共有メモリにあることをセマフォやメッセージキューを使って（トークンとして）ほかのプロセスに知らせる方法がよいでしょう。

メッセージキューのインタフェースを使えば、第11章で作成したバージョンが抱えていた問題、つまりデータを受け渡しするためには送信側と受信側の両方がパイプをオープンしていなければならないという問題を回避できます。メッセージキューでは、メッセージの送信先のプロセスが現在キューのユーザーになっていなくても、送信元のプロセスはキューにメッセージを置くことができるからです。

ただし、クライアントに応答を返す方法は検討を要します。ひとつのやり方として、サーバー用に1つ、各クライアント用に1つのメッセージキューを用意する方法があります。ただ、この方法では、同時に多数のクライアントが存在する場合に大量のメッセージキューが必要になり、問題が起きる可能性があります。一方、メッセージ内にメッセージIDフィールドを用意すれば、すべてのクライアントが1つのメッセージキューを使う場合でも、クライアントのプロセスIDをメッセージに埋め込むことで特定のクライアントプロセスに応答メッセージを渡すことができます。各クライアントは、自分に宛てられたメッセージだけを取り出し、ほかのメッセージはキューに残しておけばよいのです。

CDデータベースアプリケーションでIPC機能を使うようにするには、`pipe_imp.c`を置き換えるだけです。以下では、置き換え後のファイル`ipc_imp.c`の主要部分について説明します。

例題 サーバー関数

1 まず、必要なヘッダーファイルをインクルードし、メッセージキューのキー、およびメッセージのデータを格納する構造体を定義します。

```
#include "cd_data.h"
#include "cliserv.h"

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define SERVER_MQUEUE 1234
#define CLIENT_MQUEUE 4321

struct msg_passed {
    long int msg_key; /* used for client pid */
    message_db_t real_message;
};
```

2 msggetから返されるキュー識別子を格納するために、ファイルスコープの変数を2つ用意します。

```
static int serv_qid = -1;
static int cli_qid = -1;
```

3 メッセージキューはいずれもサーバー側で作成します。

```
int server_starting(void)
{
    #if DEBUG_TRACE
        printf("%d :- server_starting()\n",  getpid());
    #endif

    serv_qid = msgget((key_t)SERVER_MQUEUE, 0666 | IPC_CREAT);
    if (serv_qid == -1) return(0);

    cli_qid = msgget((key_t)CLIENT_MQUEUE, 0666 | IPC_CREAT);
    if (cli_qid == -1) return(0);

    return(1);
}
```

4 終了時の後始末もサーバー側で行います。サーバーが終了する際、ファイルスコープの変数にキュー識別子としては無効な値を設定し、server_endingが呼び出されたあとにサーバーがメッセージを送信しようとしても問題が起きないようにします。

```
void server_ending(void)
{
    #if DEBUG_TRACE
        printf("%d :- server_ending()\n",  getpid());
    #endif

    (void)msgctl(serv_qid, IPC_RMID, 0);
    (void)msgctl(cli_qid, IPC_RMID, 0);

    serv_qid = -1;
    cli_qid = -1;
}
```

5 サーバー側で要求を読み取る関数では、任意の型のメッセージ（つまり任意のクライアントからのメッセージ）をキューから取り出し、メッセージ内のデータの部分を（型を無視して）返します。

```
int read_request_from_client(message_db_t *rec_ptr)
{
    struct msg_passed my_msg;
    #if DEBUG_TRACE
        printf("%d :- read_request_from_client()\n",  getpid());
    #endif

    if (msgrcv(serv_qid, (void *)&my_msg, sizeof(*rec_ptr), 0, 0) == -1) {
        return(0);
    }
    *rec_ptr = my_msg.real_message;
    return(1);
}
```

6 応答を送信するときには、要求内に格納されていたクライアントのプロセスIDを使ってメッセージの宛先を指定します。

```
int send_resp_to_client(const message_db_t mess_to_send)
{
    struct msg_passed my_msg;
    #if DEBUG_TRACE
        printf("%d :- send_resp_to_client()\n",  getpid());
    #endif

    my_msg.real_message = mess_to_send;
    my_msg.msg_key = mess_to_send.client_pid;

    if (msgsnd(cli_qid, (void *)&my_msg, sizeof(mess_to_send), 0) == -1) {
        return(0);
    }
    return(1);
}
```

例題 クライアント関数

1 クライアントは、起動後にサーバーとクライアントの両方のキュー識別子を取得する必要があります。サーバーが実行されていない場合、メッセージキューは存在しないので、client_starting関数は失敗します。

```
int client_starting(void)
{
    #if DEBUG_TRACE
        printf("%d :- client_starting\n",  getpid());
    #endif

    serv_qid = msgget((key_t)SERVER_MQUEUE, 0666);
    if (serv_qid == -1) return(0);

    cli_qid = msgget((key_t)CLIENT_MQUEUE, 0666);
    if (cli_qid == -1) return(0);
    return(1);
}
```

2 サーバー同様、クライアントが終了する際、ファイルスコープの変数にキュー識別子としては無効な値を設定します。client_endingが呼び出されたあとにクライアントがメッセージを送信しようとしても問題が起きないようにします。

```
void client_ending(void)
{
    #if DEBUG_TRACE
        printf("%d :- client_ending()\n",  getpid());
    #endif

    serv_qid = -1;
    cli_qid = -1;
}
```

3 サーバーにメッセージを送信するには、構造体の内部にデータを格納します。メッセージキーも設定する必要があります。0はキーの値として不正です。未定義のままにしておくと、キーがランダムな値を取る可能性があり、たまたま値が0になっていた場合には関数が失敗します。

```
int send_mess_to_server(message_db_t mess_to_send)
{
    struct msg_passed my_msg;
    #if DEBUG_TRACE
        printf("%d :- send_mess_to_server()\n",  getpid());
    #endif

    my_msg.real_message = mess_to_send;
```

```
    my_msg.msg_key = mess_to_send.client_pid;

    if (msgsnd(serv_qid, (void *)&my_msg, sizeof(mess_to_send), 0) == -1) {
        perror("Woops");
        return(0);
    }
    return(1);
}
```

4 サーバーからメッセージを取得する際、クライアントは自分のプロセスIDを使って自分宛てのメッセージだけを取り出します。ほかのクライアント宛てのメッセージは無視します。

```
int read_resp_from_server(message_db_t *rec_ptr)
{
    struct msg_passed my_msg;
    #if DEBUG_TRACE
        printf("%d :- read_resp_from_server()\n",  getpid());
    #endif

    if (msgrcv(cli_qid, (void *)&my_msg, sizeof(*rec_ptr), getpid(), 0) == -1) {
        return(0);
    }
    *rec_ptr = my_msg.real_message;
    return(1);
}
```

5 pipe_imp.cとの完全な互換性を確保するために、以下の4つの関数を定義します。ただし、パイプを使う場合には必要だった処理もメッセージキューでは不要なので、関数の中身は空です。

```
int start_resp_to_client(const message_db_t mess_to_send)
{
    return(1);
}

void end_resp_to_client(void)
{
}

int start_resp_from_server(void)
{
    return(1);
}

void end_resp_from_server(void)
{
}
```

パイプを使うバージョンとメッセージキューを使うバージョンとを比較すると、IPCのメッセ

ージキューの利点がよくわかります。必要な関数の数が減るだけでなく、各関数も簡潔に記述することができます。

12.5 IPCステータスコマンド

X/Open仕様では必須ではありませんが、ほとんどのUNIXシステムでは、コマンドラインからIPC情報にアクセスするためのコマンド、**ipcs**と**ipcrm**が用意されています。これらのコマンドは、プログラムの開発時に使うと便利です。

12.5.1 セマフォ

システムのセマフォの状態を調べるには、**ipcs -s**コマンドを使います。セマフォが存在する場合には、次のように出力されます。

```
$ ipcs -s

------ Semaphore Arrays --------
key        semid      owner      perms      nsems      status
0x000004d2 129         rick       666        1
```

プログラムが（意図に反して）残したセマフォは、**ipcrm**コマンドで削除することができます。Linux上で上のセマフォを削除するには、次のコマンドを実行します。

```
$ ipcrm sem 129
```

その他の多くのUNIXシステムでは、次のような書式が使われています。

```
$ ipcrm -s 129
```

12.5.2 共有メモリ

セマフォ同様、コマンドラインから共有メモリの情報にアクセスすることもできます。使用するコマンドは**ipcs -m**と**ipcrm shm <id>**（または**ipcrm -m <id>**）です。

次に示すのは、**ipcs -m**コマンドからの出力例です。

```
$ ipcs -m

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x000004d2 130        rick       666        4096       2
```

4KBの1つの共有メモリセグメントが2つのプロセスによってアタッチされていることがわかります。

ipcrm shm <id>コマンドを使うと、共有メモリを削除することができます。プログラムが共有メモリ使用後の後始末に失敗した場合などに使用します。

12.5.3 メッセージキュー

メッセージキューの場合には、**ipcs -q**と**ipcrm msg <id>**（または**ipcrm -q <id>**）を使います。

次に示すのは**ipcs -q**からの出力例です

```
$ ipcs -q

------ Message Queues --------
key        msqid      owner      perms      used-bytes  messages
0x000004d2 257         rick      666        3072        3
```

メッセージキューに3つのメッセージがあり、合計サイズが3072バイトであることがわかります。

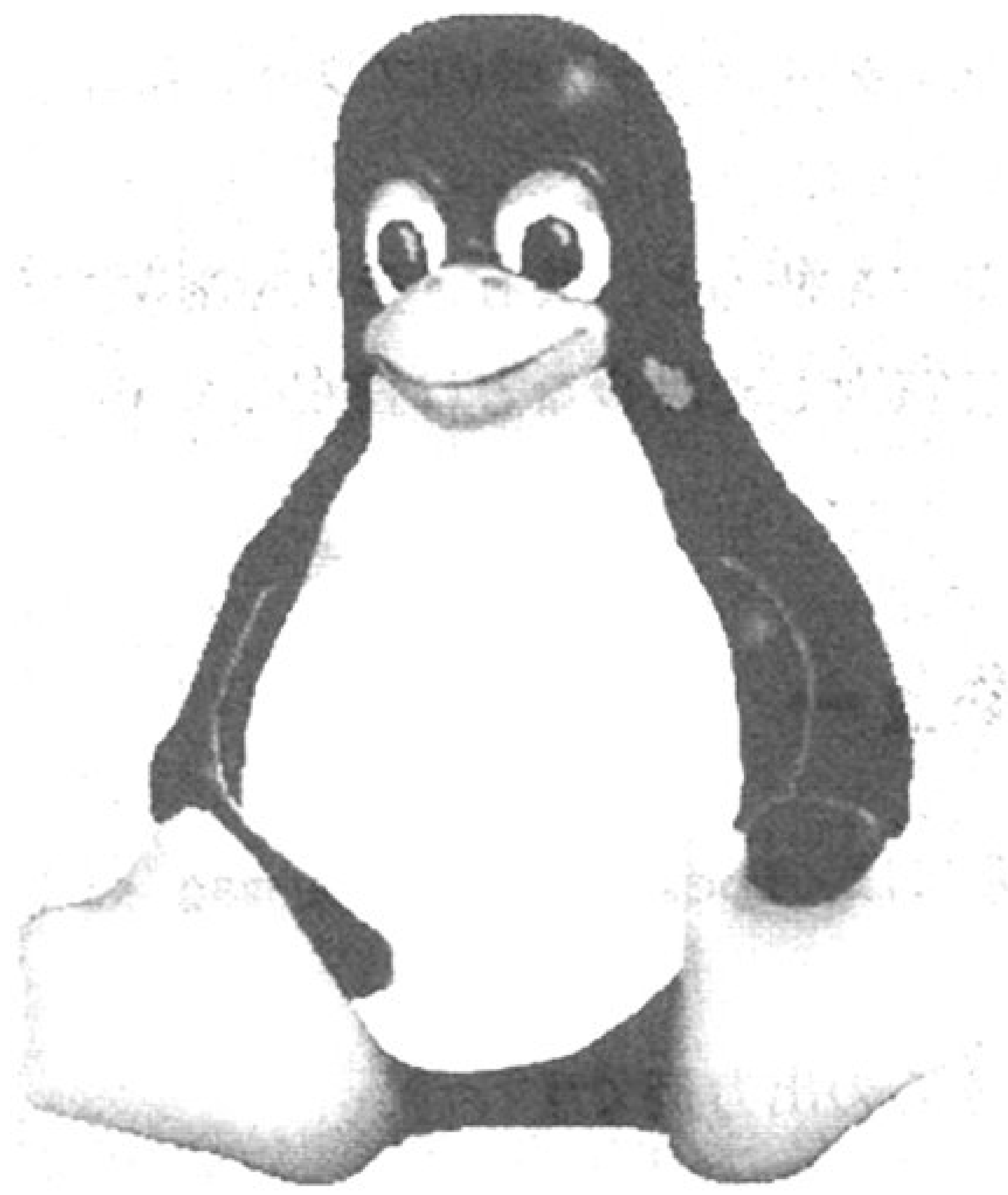
メッセージキューを削除するには、**ipcrm msg <id>**コマンドを使います。

12.6 この章のまとめ

この章では、UNIX System V.2で導入されたセマフォ、共有メモリ、メッセージキューの3つのプロセス間通信機能について取り上げました。System V IPCが提供する高度な機能を使うと、プロセス間通信を行うプログラムが抱えるさまざまな課題にも柔軟に対処することができます。

注意

UNIXという呼称を使用するためにはどのような機能を提供しなければならないかを定めたX/Open仕様では、この章で取り上げたIPC関連関数についてコメントが付けられています。このコメントでは、IEEE 1003.4によってプロセス間通信のための代替インタフェースが現在開発されており、IPC機能を使ったアプリケーションを作成する際は、今後代替インタフェースを使うようになった場合でも修正しやすいように設計すべきである、と述べられています。IPC機能を使う場合にはこの点にも注意しておくとよいでしょう。



第13章 ソケット

この章では、もうひとつのプロセス間通信の方法について取り上げます。これまでに取り上げた方法は、1台のコンピュータ上で共有リソースを使うことを前提としたものでした。この場合、使用するリソースは、ファイルシステム空間や共有メモリ、メッセージキューなど、それぞれの方法ごとに異なりますが、リソースを使用できるのは該当するコンピュータで実行中のプロセスだけに限られています。

Berkeleyバージョンの UNIXは、パイプの概念を拡張した**ソケットインタフェース**と呼ばれる新しい通信手段を導入しました。ソケットはパイプとほぼ同様に使用できますが、ネットワークを介した通信も扱うことができるようになっており、汎用性がいっそう高まっています。具体的には、あるコンピュータ上のプロセスと別のコンピュータ上のプロセスとが、ソケットを使うことで互いに通信できるようになっています。ソケットを利用すると、クライアントサーバーシステムをネットワーク上に分散して配置することができます。

ソケットインタフェースに関しては**Windows Sockets**、略して`WinSock`と呼ばれる公開仕様があり、Microsoft Windowsでもソケットを利用できるようになっています。Windowsのソケットサービスは`Winsock.dll`というシステムファイルによって提供されており、ネットワークを介してUNIXコンピュータと通信するWindowsプログラムを作成すれば、異なるOS間でクライアントサーバーシステムを構築することも可能です。`WinSock`のプログラミングインタフェースはUNIXのソケットとはやや異なりますが、コンセプト自体は同じです。

UNIXには実にさまざまなネットワーク機能があり、この章だけでそのすべてについて詳しく解説することはできません。ここでは、自分でネットワークプログラムを作成できるようになることを目標に、プログラミングに関係のあるネットワークインタフェースについて説明します。取り上げる項目は次のとおりです。

- ソケット接続の動作のしくみ
- ソケットの属性、アドレス、通信
- ネットワーク情報とインターネットデーモン
- クライアントとサーバー

13.1 ソケットとは

ソケットは、1台のマシン上でのローカルなクライアントサーバーシステムや、ネットワークを介したクライアントサーバーシステムの開発を可能にする通信機構です。UNIXが提供する印刷などの機能、`rlogin`や`ftp`といったネットワークユーティリティの通信機能は、ソケットを使って実現されています。

ソケットではクライアントとサーバーが明確に区別されるので、作成方法や使い方もパイプと

は異なります。ソケットでは、1つのサーバーに対して複数のクライアントが接続するシステムも比較的容易に実装することができます。

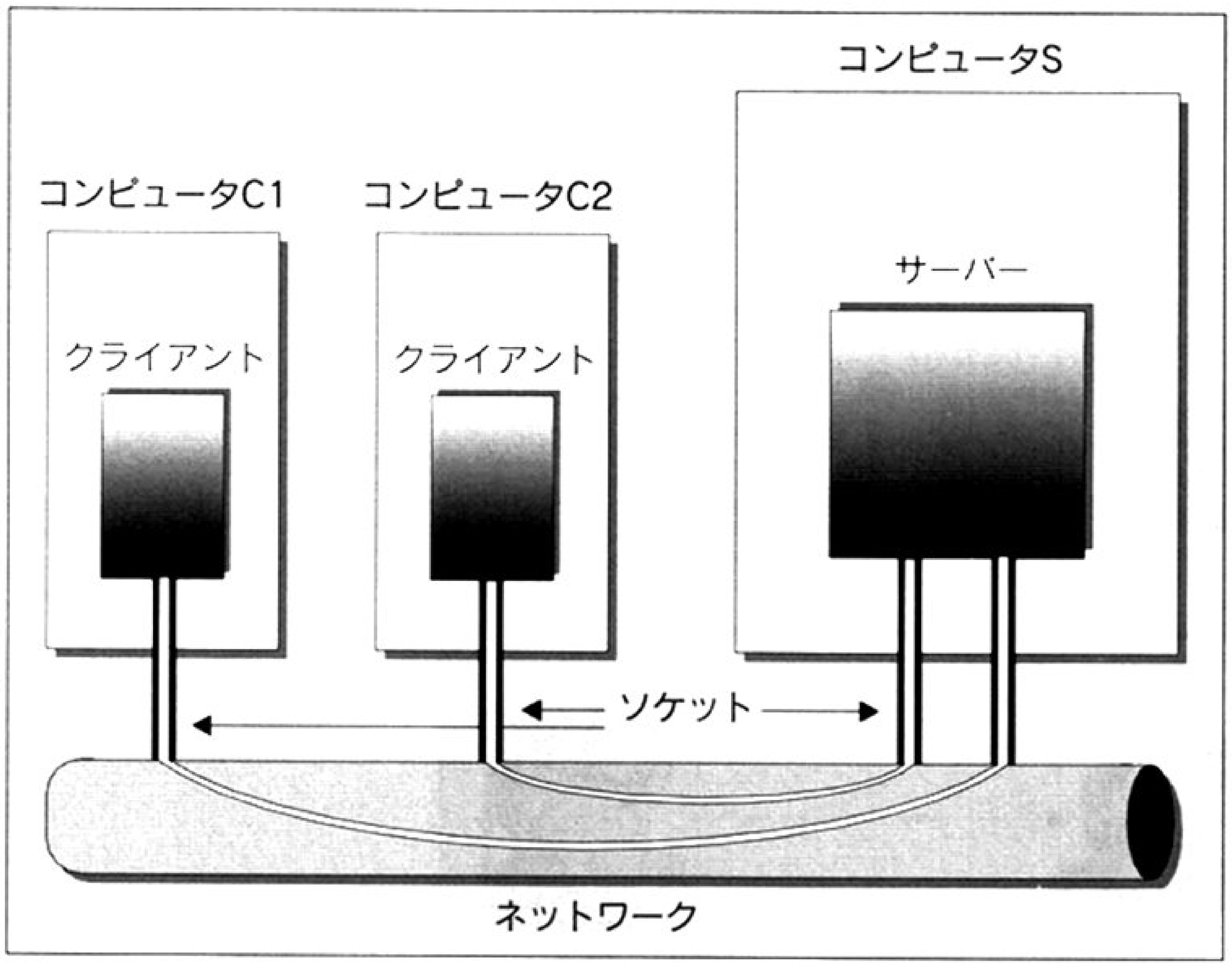


図13.1 ソケットネットワークの構成

13.2 ソケット接続

ソケット接続の働きは、ひっきりなしに電話がかかってくる会社にたとえて考えると理解しやすいかもしれません。会社にかかってきた電話は受付によって適切な部署にまわされ（サーバープロセス）、さらにそこから適切な担当者（サーバーソケット）へと転送されます。この時点で、かかってきた電話（クライアント）と適切な通話先（エンドポイント）とが結び付けられたことになり、間に介在するオペレータ役（受付や部長など）はほかの電話を受けることができるようになります。では、UNIXはこうしたソケット接続をどのような方法で確立するのでしょうか。

まず最初に、サーバーアプリケーションがソケットを作成します。これは、サーバープロセスに割り当てられるオペレーティングシステムのリソースです。サーバーが`socket`システムコールを使ってソケットを作成した時点では、ほかのプロセスはまだソケットにアクセスできません。

次に、サーバープロセスはソケットに名前を付けます。ローカルソケットには、UNIXファイルシステムのファイル名が与えられます（`/tmp`や`/usr/tmp`には、しばしばこうしたファイルがあります）。一方、ネットワークソケットの場合には、クライアントから接続可能な特定のネットワークに応じたサービス識別子（ポート番号／アクセスポイント）が名前になります。ソケッ

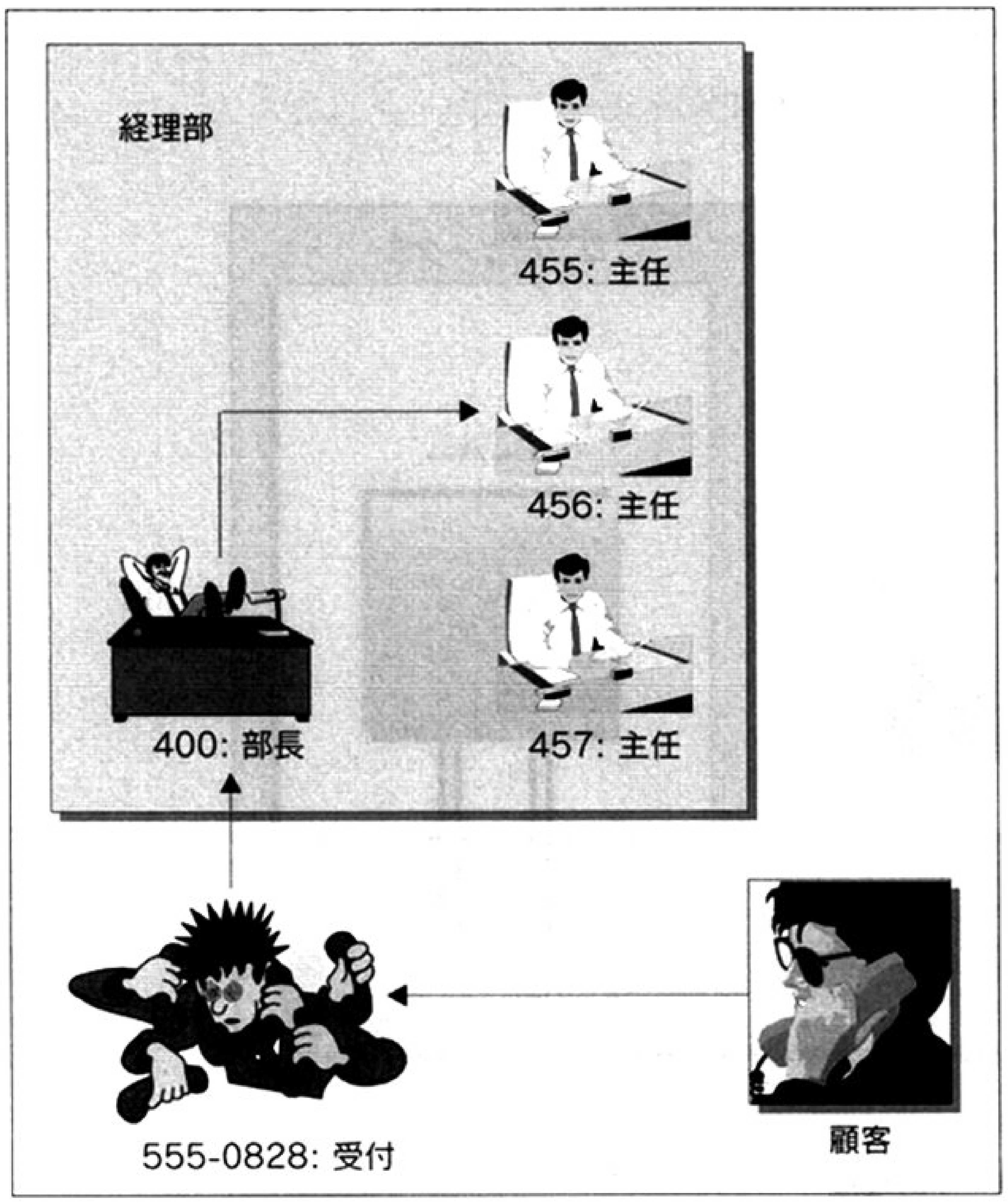


図13.2 クライアントがサーバープロセスを介してエンドポイントにアクセスする

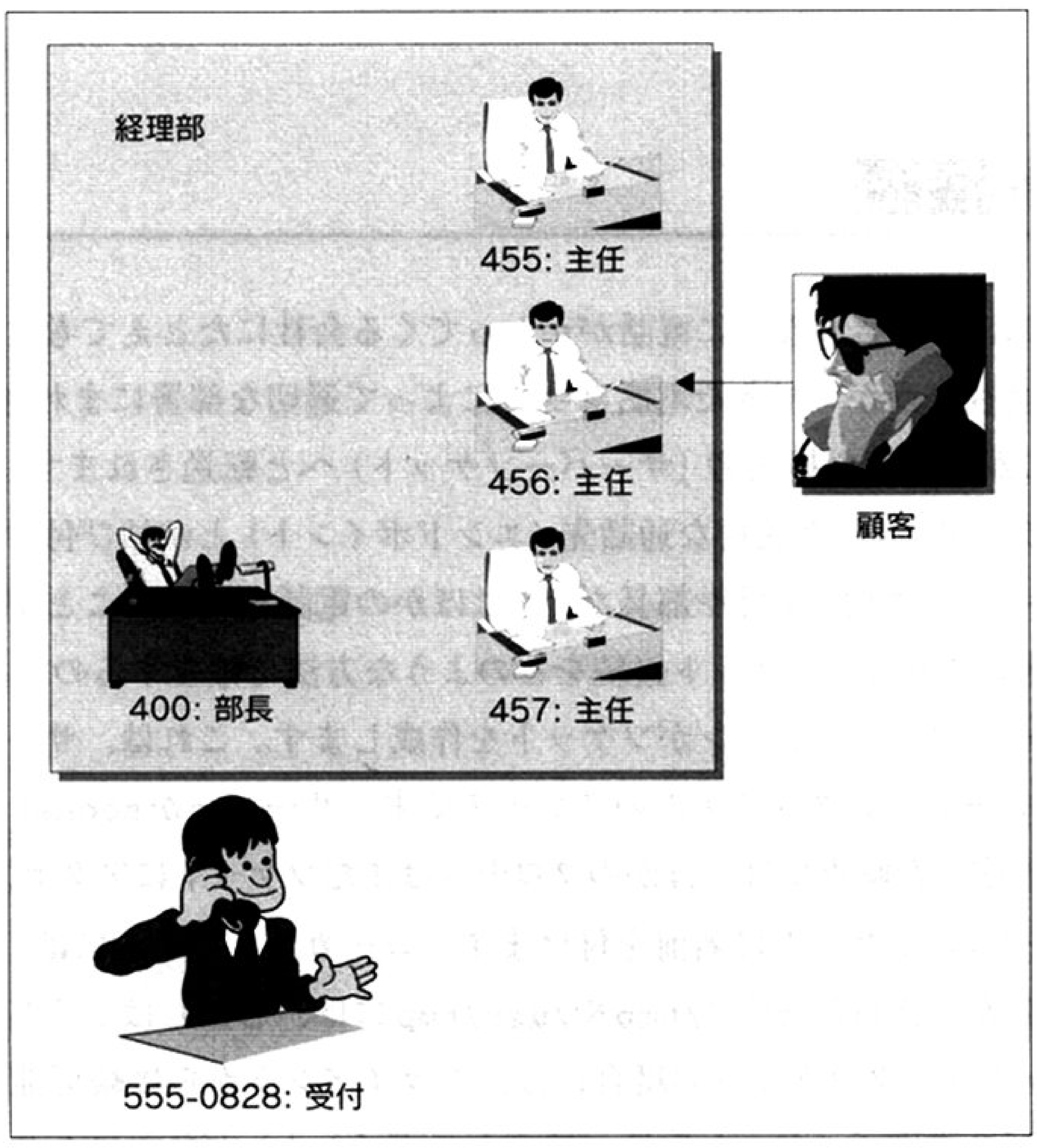


図13.3 クライアントがエンドポイントと接続する

トに名前を付けるには、bindシステムコールを使います。

サーバープロセスは、名前を付けたソケットにクライアントが接続してくるのを待ちます。また、同時に複数のクライアントが接続してきた場合のために、listenシステムコールによって接続用のキューを作成します。サーバーは、acceptシステムコールによって接続を受け入れることができます。

サーバーがacceptを呼び出すと、名前の付いたソケットとは別の新しいソケットが作成されます。この新しいソケットは、接続してきた特定のクライアントと通信するためだけに使われます。名前の付いたソケットはそのまま残っており、ほかのクライアントからの接続用に利用することができます。このため、サーバー側を適切に記述すれば複数の接続を扱うことができます。単純なサーバーの場合、あとから接続してきたクライアントは、サーバーが対応できるようになるまでリスンキューで待機します。

ソケットベースのシステムでは、クライアント側の手順は簡単です。クライアントは、socketを呼び出して名前のないソケットを作成します。次に、connectを呼び出し、名前の付いたソケットをアドレスに使ってサーバーとの接続を確立します。

接続の確立後は、ソケットを低水準ファイルデスクリプタのように扱うことができ、双方向のデータ通信が可能になります。

次に示すclient1.cは、ごく簡単なソケットクライアントプログラムです。名前のないソケットを作成し、server_socketという名前のサーバーソケットに接続します。socketシステムコールについては、少しあとで詳しく説明します。

例題 簡単なローカルクライアント

1 ヘッダーファイルをインクルードし、必要な変数を用意します。

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <sys/un.h>
#include <unistd.h>

int main()
{
    int sockfd;
    int len;
    struct sockaddr_un address;
    int result;
    char ch = 'A';
```

2 クライアント用ソケットを作成します。

```
    sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
```

3 サーバー側と同じ名前でソケットの名前を指定します。

```
address.sun_family = AF_UNIX;
strcpy(address.sun_path, "server_socket");
len = sizeof(address);
```

4 クライアントのソケットをサーバーのソケットに接続します。

```
result = connect(sockfd, (struct sockaddr *)&address, len);

if(result == -1) {
    perror("oops: client1");
    exit(1);
}
```

5 sockfdを介して読み書きができるようになります。

```
    write(sockfd, &ch, 1);
    read(sockfd, &ch, 1);
    printf("char from server = %c\n", ch);
    close(sockfd);
    exit(0);
}
```

このプログラムを実行するとエラーになります。これは、接続先のサーバー側のソケットをまだ作成していないためです。

```
$ client1
oops: client1: No such file or directory
$
```

次に示すserver1.cは、クライアントからの接続を受け付ける簡単なサーバープログラムです。サーバーソケットを作成し、ソケットに名前を付け、リスンキューを作成し、接続を受け入れます。

例題 簡単なローカルサーバー

1 ヘッダーファイルをインクルードし、必要な変数を用意します。

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <sys/un.h>
#include <unistd.h>

int main()
```

```
{
    int server_sockfd, client_sockfd;
    int server_len, client_len;
    struct sockaddr_un server_address;
    struct sockaddr_un client_address;
```

2 古いソケットが残っていれば削除し、サーバー用に名前のないソケットを作成します。

```
    unlink("server_socket");
    server_sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
```

3 bindを呼び出してソケットに名前を付けます。

```
    server_address.sun_family = AF_UNIX;
    strcpy(server_address.sun_path, "server_socket");
    server_len = sizeof(server_address);
    bind(server_sockfd, (struct sockaddr *)&server_address, server_len);
```

4 接続キューを作成し、クライアントからの接続を待ちます。

```
    listen(server_sockfd, 5);
    while(1) {
        char ch;

        printf("server waiting\n");
```

5 接続を受け入れます。

```
        client_sockfd = accept(server_sockfd,
            (struct sockaddr *)&client_address, &client_len);
```

6 client_sockfdを介してクライアントに対する読み書きができるようになります。

```
        read(client_sockfd, &ch, 1);
        ch++;
        write(client_sockfd, &ch, 1);
        close(client_sockfd);
    }
}
```

このサーバープログラムでは、同時に1つのクライアントに対してのみサービスを提供します。サービスの内容は、クライアントから1文字を読み取り、これをインクリメントしてから返すというものです。クライアントに代わってもっと高度な処理を行う場合、同時に1つのクライアントしか扱うことのできないサーバーでは役不足です。ほかのクライアントは、サーバーが処理を終えるまで接続できないからです。同時に複数の接続を可能にする方法についてはあとで取り上げます。

13

このプログラムを実行すると、サーバーはソケットを作成して接続を待ちます。サーバープログラムをバックグラウンドで実行しておけば、クライアントをフォアグラウンドで起動することができます。

```
$ server1 &
[1] 1094
$ server waiting
```

接続を待つ際、サーバーはメッセージを表示します。このサンプルプログラムの場合、サーバーが接続を待っているソケットはファイルシステムソケットです。該当するファイルの存在は、**ls**コマンドで確かめることができます。一般に、ソケットを使い終わったら（プログラムがシグナルを受け取って終了した場合も含めて）、ソケットを削除しておきます。

```
$ ls -lF server_socket
srwxr-xr-x   1 neil     users           0 Jan 14 08:28 server_socket=
```

パーミッションの先頭の**s**とファイル名の最後の**=**は、デバイスタイプがソケットであることを示しています。ソケットは、ふつうのファイルと同様に作成され、パーミッションも現在の**umask**によって修正されます。**ps**コマンドを使うと、サーバーがバックグラウンドで実行中であることを確かめることができます。**STAT**列には**S**が表示され、プロセスが現在休眠中で、CPUリソースを消費していないことがわかります。

```
$ ps -lx
 FLAGS   UID   PID  PPID PRI  NI   SIZE   RSS WCHAN       STA TTY TIME COMMAND
100000   502 22727 20453  11   0    808   228  13bd7a     S   p4  0:00 server1
```

サーバープログラムが実行されていることを確認したところで、クライアントプログラムを実行してみましょう。今度はサーバーソケットが存在するので、ソケットに接続してサーバーと通信することができます。

```
$ client1
server waiting
char from server = B
$
```

端末上では、クライアントとサーバーからの出力が一緒になって表示されていますが、サーバーがクライアントから1文字を受け取り、これをインクリメントしてから返していることがわかります。サーバーは実行を続け、次のクライアントが接続してくるのを待ちます。クライアントをいくつか同時に実行すると、各クライアントは順番にサーバーからサービスを受けます。

```
$ client1 & client1 & client1 &
[2] 1106
[3] 1107
[4] 1108
server waiting
char from server = B
server waiting
char from server = B
server waiting
char from server = B
server waiting
[2]   Done                    client1
[3]-  Done                    client1
[4]+  Done                    client1
$
```

13.2.1 ソケットの属性

サンプルプログラムで使っているシステムコールについて理解するには、UNIXのネットワークについて少し知っておく必要があります。

ソケットは、ドメイン、タイプ、プロトコルという3つの属性によって特徴付けられます（それぞれの属性については以下で順に説明していきます）。また、ソケットはアドレスを持ち、このアドレスがソケットの名前として使われます。アドレスの形式はドメインによって異なります。ドメインはプロトコルファミリと呼ばれることもあります。各プロトコルファミリでは、アドレスの形式を定義するのに1つまたは複数のアドレスファミリを使用できます。

◆ ソケットのドメイン

ドメインは、ソケット通信で使用するネットワーク媒体を指定するものです。最もよく使われるソケットドメインは、インターネットネットワークを意味する**AF_INET**です。インターネットネットワークは、多くのUNIX LAN、そしてインターネットで使われています。インターネットネットワークを支えているプロトコル、いわゆる**インターネットプロトコル（IP）**は、1つのアドレスファミリだけを持ち、ネットワーク上のコンピュータを指定するのに特定の方法を用います。これが**IPアドレス**です。

ほとんどの場合、インターネット上のマシンを特定するには名前を使いますが、これらの名前は、実際に通信を行う前に低水準のIPアドレスに変換されます。すべてのIPアドレスは、192.168.1.99のように、それぞれが256より小さい4つの数で表されます（この表記法のことを10進数ドット表記といいます）。クライアントがネットワーク上でソケットを介してサーバーと接続するには、サーバーコンピュータのIPアドレスを知っている（または見つける）必要があります。

一方、サーバー上では複数のサービスが提供されている可能性があります。そこで、クライア

ントはIPポートを使って接続先マシンの特定のサービスを指定します。システム内部では、一意の16ビット整数を割り当てることによってポートを識別し、外部との間では、IPアドレスとポート番号の組み合わせによってポートを識別します。ソケットは通信の端点（エンドポイント）であり、実際に通信を行う前にソケットをポートに結び付けなければ（バインドしなければ）なりません。

サーバー側では、特定のポートで接続を待ちます。よく知られたサービス（既知のサービス）には、あらかじめ決められたポート番号が割り当てられており、これらのポート番号はすべてのUNIXマシンで共通に使われます。この種のポート番号は、1024より小さいのがふつうです。プリンタスプーラ（515）、`rlogin`（513）、`ftp`（21）、`http`（80）などはその例です。ちなみに最後の`http`は、WWWサーバーが使うポートです。一般に、1024より小さいポートはシステムサービスのために予約されており、スーパーユーザー特権を持つプロセスだけが使用できます。X/Open仕様では、最高位の予約ポート番号を表す定数`IPPORT_RESERVED`を`netdb.h`で定義しています。

標準的なサービスについては一連のポート番号が定まっているので、コンピュータはこれらのポート番号を使って互いに容易に接続することができます。一方、ローカルなサービスでは、非標準的なポートアドレスを使うことがあります。すでにサンプルプログラムで使ったドメインは、UNIXファイルシステムのドメイン`AF_UNIX`です。`AF_UNIX`は、ネットワークに接続されていない1台のコンピュータ上でも使用できるドメインで、そのプロトコルはファイル入出力であり、アドレスは絶対ファイル名です。サンプルプログラムではサーバーソケットのアドレスとして`server_socket`を使いましたが、これがサーバーアプリケーションを実行したときに現在のディレクトリ内のファイルとして存在することは、すでに見たとおりです。

`AF_INET`や`AF_UNIX`以外のドメインとしては、ISO標準プロトコルに基づくネットワークを表す`AF_ISO`、Xerox Network Systemを表す`AF_NS`などがありますが、一般的ではないので本書では取り上げません。

◆ ソケットのタイプ

1つのソケットドメインには、それぞれ特徴の異なるさまざまな通信手段が用意されていることがあります。`AF_UNIX`ドメインソケットでは、この点が問題になることはありません。信頼性の高い双方向の通信経路を利用できるからです。しかし、ネットワークドメインの場合には、ネットワークの特性に注意を向ける必要があります。

インターネットプロトコルは、**ストリーム**と**データグラム**という2つの異なるレベルのサービスを提供しています。

このうち、ストリーム（いくつかの点で標準入出力ストリームと似ています）では、信頼性の高い順次双方向のバイトストリームに基づく接続を利用できます。かみくだいていえば、送信されたデータが失われたり、重複したり、順序が入れ替わったりすることのないことが保証されており、もしそうした状況が発生した場合には、エラーによってそれとわかるという意味です。大

きなメッセージはフラグメント化されて転送され、あとで組み立て直されます。これは、大量のデータを受け取ったファイルストリームが、データを小さなブロックに分けて低水準ディスクに書き込みを行うのとよく似ています。ストリームソケットでは動作の予測が可能です。

ストリームソケットは、**SOCK_STREAM**というタイプによって指定し、**AF_INET**ドメインではTCP/IP接続によって実装されています。また、ストリームソケットは**AF_UNIX**ドメインでもよく使われます。**SOCK_STREAM**ソケットはネットワークアプリケーションのプログラミングでは一般的なので、この章でも主にストリームソケットについて解説します。

メモ

TCP/IPは、Transmission Control Protocol/Internet Protocol（伝送制御プロトコル/インターネットプロトコル）の略です。TCP/IPは、ネットワークを介したコンピュータからコンピュータへのルーティングをサポートしています。大量の転送データが確実に目的地に到着することを保証するために、TCP/IPでは順序付け、フロー制御、再送が可能になっています。

一方、**SOCK_DGRAM**というタイプによって指定するデータグラムソケットは、接続の確立と維持を行いません。また、送信できるデータグラムのサイズにも制限があります。データグラムは単独のネットワークメッセージとして送信され、途中で失われたり、重複したり、順序が入れ替わったりして目的地に到着することがあります。

データグラムソケットは、**AF_INET**ドメインではUDP/IP接続によって実装されており、送信された順に到着するとは限らない信頼性の低いサービスを提供します。ただし、使用するリソースの観点から見れば、ネットワーク接続を維持する必要がない分、データグラムソケットのほうが安価に利用できます。また、接続をセットアップする時間も必要なく、高速です。UDPはUser Datagram Protocol（ユーザーデータグラムプロトコル）の略です。

情報サービスに対する1回きりの問い合わせや、ステータス情報の定期的な提供、プライオリティの低いログの収集などを行う場合には、データグラムが適しています。サーバーが“死んだ”場合にクライアントの再起動が必要ないことも利点です。

◆ ソケットのプロトコル

要求されたソケットタイプを提供するのに複数のプロトコルを使用できる場合には、特定のプロトコルを選択することができます。この章では、UNIXネットワークとファイルシステムのソケットだけを取り上げるので、（デフォルト以外の）プロトコルを選択する必要はありません。

13.3 ソケットの使い方

ソケットの背景についてはひととおり説明が済んだので、以下でソケット関連のシステムコールについて詳しく解説しましょう。

13.3.1 ソケットの作成

socketシステムコールはソケットを作成し、ソケットへのアクセスに使用できるデスクリプタを返します。

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

作成されたソケットは、通信チャネルの一方の端点になります。パラメータ**domain**にはアドレスファミリを指定し、**type**にはソケットで使う通信のタイプを指定します。パラメータ**protocol**には使用するプロトコルを指定します。

ドメインには次のようなものがあります。

- **AF_UNIX**　UNIX内部（ファイルシステムソケット）
- **AF_INET**　ARPAインターネットプロトコル（UNIXネットワークソケット）
- **AF_ISO**　ISO標準プロトコル
- **AF_NS**　Xerox Network Systemsプロトコル

最もよく使われるソケットドメインは、**AF_UNIX**と**AF_INET**です。**AF_UNIX**は、UNIXファイルシステムを介して実装されるローカルソケットを使うときに指定します。**AF_INET**はUNIXネットワークソケット用のソケットドメインで、インターネットをはじめ、TCP/IPネットワーク経由で通信するプログラムで使うことができます。Microsoft WindowsのWinSockインタフェースでもこのソケットドメインにアクセスすることができます。

パラメータ**type**は、新しく作成するソケットで使う通信のタイプを表し、次の値を指定することができます。

- **SOCK_STREAM**
- **SOCK_DGRAM**

SOCK_STREAMは、信頼性の高い順次双方向のバイトストリームです。**AF_INET**ドメインソケットの場合、**SOCK_STREAM**はTCP接続によって提供されます。TCP接続は、ストリームソケットの接続時に2端点間で確立されます。データは、ソケット接続を介して双方向にやりとりすることができます。TCPプロトコルには、長いメッセージをフラグメント化して再び組み立て直す機能が備わっているほか、ネットワーク経路の途上で失われた任意の部分を再送することもできます。

SOCK_DGRAMはデータグラムサービスです。このソケットを使うと、固定最大長サイズ（一般には小さいサイズ）のメッセージを送信することができますが、メッセージが目的地に配送されることも、ネットワーク上で順序が入れ替わらないことも保証されません。**AF_INET**ドメインソケットの場合、**SOCK_DGRAM**はUDPデータグラムによって提供されます。

通信に使うプロトコルは、一般にソケットのタイプとドメインによって決まります。通常、選択肢は1つだけです。**socket**システムコールのパラメータ**protocol**は、複数のプロトコルファミリから1つを選択する必要がある場合に、特定のプロトコルを指定するのに使います。0を指定すると、デフォルトのプロトコルが選択されます。この章で取り上げる例では、いずれもこのパラメータには0を指定します。

socketシステムコールはデスクリプタを返します。このデスクリプタは、多くの点で低水準ファイルデスクリプタによく似ています。たとえば、ソケットがいったんもう一方のソケットと接続されたあとは、**socket**システムコールから返されたデスクリプタを指定し、**read**システムコールと**write**システムコールを使ってデータのやりとりを行うことができます。また、ソケット接続を終了するのにも**close**システムコールを使います。

13.3.2 ソケットのアドレス

各ソケットドメインには独自のアドレス形式が必要です。**AF_UNIX**ソケットの場合、アドレスは、**sys/un.h**で定義されている次のような構造体**sockaddr_un**で表現されます。

```
struct sockaddr_un {
    sa_family_t   sun_family;    /* AF_UNIX */
    char          sun_path[];    /* pathname */
};
```

ソケットを扱うシステムコールに対してさまざまなタイプのアドレスを渡すことができるようにするために、各アドレスの形式は上と同じような構造体で表されます。具体的には、構造体の先頭（**sockaddr_un**の場合には**sun_family**）に、アドレスのタイプ（ソケットドメイン）を表すフィールドがあります。**AF_UNIX**ドメインの場合、アドレスは**sun_path**フィールドのファイル名によって表されます。

現在のLinuxシステムでは、**sa_family_t**型は**short**です（**sa_family_t**型は、**sys/un.h**

で宣言すべきものとしてX/Open仕様で定義されています)。また、**sun_path**で指定される**pathname**の長さは、**UNIX_MAX_PATH**文字に制限されています。アドレス構造体はサイズが変わるので、多くのソケットシステムコールではアドレスのサイズを指定する必要があるか、または出力にサイズが含まれています。このサイズは、対象となるアドレス構造体をコピーするときに利用することができます。

AF_INETドメインでは、**netinet.h**で定義されている**sockaddr_in**構造体を使ってアドレスを指定します。**sockaddr_in**構造体には、少なくとも次のメンバがあります。

```
struct sockaddr_in {
    short int            sin_family;  /* AF_INET */
    unsigned short int   sin_port;    /* Port number */
    struct in_addr       sin_addr;    /* Internet address */
};
```

IPアドレス構造体は次のように定義されています。

```
struct in_addr {
    unsigned long int     s_addr;
};
```

この構造体では、IPアドレスの4つのバイトが1つの32ビット値で表されます。**AF_INET**ソケットを表現するには、ドメイン、IPアドレス、およびポート番号だけで十分です。アプリケーションの観点から見ると、すべてのソケットはファイルデスクリプタと同じようにふるまい、一意の整数値によって識別することができます。

13.3.3 ソケットの命名

ほかのプロセスからソケットを利用できるようにするには、サーバープロトコルがソケットに名前を付ける必要があります。すでにサンプルプログラム**server1**で示したように、**AF_UNIX**ソケットの場合には、ファイルシステムのパス名をソケットに関連付けます。**AF_INET**ソケットの場合には、ソケットをIPポート番号に関連付けます。

```
#include <sys/socket.h>

int bind(int socket, const struct sockaddr *address, size_t address_len);
```

bindシステムコールを使うと、ファイルデスクリプタ**socket**に関連付けられている名前のないソケットに対して、パラメータ**address**で指定されたアドレスを割り当てることができます。パラメータ**address_len**には、アドレス構造体の長さを指定します。

アドレスの長さと形式はアドレスファミリによって異なります。**bind**を呼び出すときには、ア

ドレス構造体へのポインタを汎用アドレス型(**struct sockaddr ***)にキャストする必要があります。

bindシステムコールは、成功すると0を返します。失敗した場合には-1を返し、**errno**には次のいずれかの値が設定されます。

- **EBADF** ファイルデスクリプタが無効である。
- **ENOTSOCK** ファイルデスクリプタがソケットを参照していない。
- **EINVAL** ファイルデスクリプタがすでに名前の付いたソケットを参照している。
- **EADDRNOTAVAIL** アドレスを利用できない。
- **EADDRINUSE** アドレスにはすでにソケットがバインドされている。

上の値に加えて、**AF_INET**ソケットではさらに次の値を取ることもあります。

- **EACCESS** 指定されたファイルシステム名を作成するためのパーミッションがない。
- **ENOTDIR**、**ENAMETOOLONG** ファイル名の指定が適切ではない。

13.3.4 ソケットキューの作成

ソケット上で接続を受け入れるには、保留中の要求を格納するためのキューをサーバープログラムが作成する必要があります。

```
#include <sys/socket.h>

int listen(int socket, int backlog);
```

キューに保持できる保留中の接続の最大数は、システムによって制限されていることがあります。**listen**は、この制限を考慮したうえでキューの長さを**backlog**に設定します。該当するソケットへの接続はこの長さまでは保留中として保持されますが、それ以上の接続は拒否され、クライアントからの接続は失敗します。**listen**が提供する機能は、サーバープログラムがクライアントの処理でビジー状態にあるときでも接続を保留中として保持できるようにするためのものです。**backlog**の値としては、5が一般的です。

listenシステムコールは成功すると0を返し、失敗した場合には-1を返します。エラーには**EBADF**、**EINVAL**、**ENOTSOCK**があり、それぞれの意味は**bind**システムコールの場合と同じです。

13.3.5 接続の受け入れ

ソケットを作成して名前を付けたあと、サーバープログラムはacceptシステムコールを使って、該当するソケットへの接続を待つことができます。

```
#include <sys/socket.h>

int accept(int socket, struct sockaddr *address, size_t *address_len);
```

acceptシステムコールは、パラメータsocketで指定されたソケットにクライアントプログラムが接続を試みると戻ります。このクライアントは、ソケットのキューのうち保留中の最初の接続です。acceptは、クライアントと通信するための新しいソケットを作成し、そのデスクリプタを返します。新しいソケットのタイプはサーバーのリスンソケットと同じです。

ソケットには、あらかじめbindによって名前が付けられ、listenによってキューが割り当てられていなければなりません。呼び出し側のクライアントのアドレスは、addressによって示されるsockaddr構造体に格納されます。クライアントのアドレスを必要としない場合には、addressにヌルポインタを指定することもできます。

パラメータaddress_lenは、アドレス構造体の長さを指定するのに使います。クライアントアドレスがこの値より長い場合、そのアドレスは切り捨てられます。呼び出しから戻ると、address_lenには呼び出し側のクライアントのアドレス構造体の実際の長さがセットされます。

ソケットのキューに保留中の接続がない場合、クライアントが接続してくるまでacceptはブロックされます。ただし、この動作は、fcntl関数を使ってソケットのファイルデスクリプタにO_NONBLOCKフラグを指定することで変更できます。具体的には次のようにします。

```
int flags = fcntl(socket, F_GETFL, 0);

fcntl(socket, F_SETFL, O_NONBLOCK|flags);
```

acceptシステムコールは、保留中のクライアント接続がある場合には新しいソケットファイルデスクリプタを返し、エラーが発生した場合には -1を返します。エラーの内容はbindやlistenと同じですが、それ以外にEWOULDBLOCKがあります。EWOULDBLOCKは、O_NONBLOCKが指定されていて保留中の接続がない場合に発生します。また、acceptでブロックされているときにプロセスが割り込まれると、EINTRが発生します。

13.3.6 接続の要求

クライアントプログラムは、connectシステムコールを呼び出し、名前のないソケットとサーバーのリスンソケットとの間の接続を確立することによってサーバーと接続します。

```
#include <sys/socket.h>

int connect(int socket, const struct sockaddr *address, size_t address_len);
```

connectは、パラメータsocketで指定されたソケットを、パラメータaddressで指定されたaddress_lenの長さを持つサーバーソケットに接続します。指定するソケットは、socketから返された有効なファイルデスクリプタでなければなりません。

connectは成功すると0を返し、失敗した場合には-1を返します。エラーには次のようなものがあります。

- **EBADF** socketに指定されたファイルデスクリプタが無効である。
- **EALREADY** 指定されたソケットで接続がすでに進行中である。
- **ETIMEDOUT** 接続タイムアウトが発生した。
- **ECONNREFUSED** 要求された接続がサーバーによって拒否された。

接続をただちにセットアップできない場合、connectは特に指定のないタイムアウト期間、ブロックされます。このタイムアウト期間が過ぎると、接続は異常終了し、connectは失敗します。ただし、connectの呼び出しが、ハンドラで処理されるシグナルによって割り込まれた場合、呼び出しは失敗しますが（errnoにはEINTRが設定される）、接続の試みは異常終了せず、非同期で接続がセットアップされます。

acceptの場合と同様、該当するファイルデスクリプタに対してO_NONBLOCKフラグを設定すると、connectの動作を変更することができます。O_NONBLOCKを指定した場合、接続をすぐにセットアップできなければconnectは失敗します。errnoにはEINPROGRESSが設定され、非同期で接続がセットアップされます。

非同期接続は扱いがやや面倒ですが、ソケットファイルデスクリプタに対してselectを呼び出すことで、ソケットに書き込みできるかどうかを調べることができます。selectについてはあとで詳しく取り上げます。

13.3.7 ソケットのクローズ

低水準ファイルデスクリプタの場合と同様、closeを呼び出すと、サーバーとクライアントのソケット接続を終了することができます。ソケットは常に両側でクローズする必要があります。サーバー側では、readが0を返したときにcloseを呼び出す必要がありますが、転送されていないデータがソケットにあり、ソケットが接続指向型で、かつSOCK_LINGERオプションが設定されている場合には、closeの呼び出しがブロックされる可能性があります。ソケットのオプションの設定についてはあとで取り上げます。

13.4 ソケット通信

以上でソケット関連のシステムコールの説明は終わりです。以下では、サンプルプログラムを使ってソケットの動作を具体的に見ていくことにしましょう。

まず、最初のサンプルプログラムを修正し、ファイルシステムソケットの代わりにネットワークソケットを使います。ファイルシステムソケットの場合には、絶対パス名を使わないかぎり、サーバープログラムの現在のディレクトリにソケットが作成されてしまうという欠点があります。ファイルシステムソケットを汎用的に利用できるようにするには、グローバルにアクセス可能な`/tmp`などのディレクトリにソケットを作成し、この場所に関してサーバーとクライアントの間であらかじめ合意が成立していなければなりません。一方、ネットワークソケットの場合には、このような点を考慮する必要はなく、未使用のポート番号を選択するだけですみます。

ここでは、ポート番号として9734を使うことにします。9734という番号に特に意味はありませんが、標準的なサービスを避けるようにしています。すでに説明したように、1024未満のポート番号はシステム用に予約されており、ユーザーが勝手に使用することはできません。また、そのほかのポート番号も（必ずしも必要ではありませんが）該当するポート番号で提供されているサービスとともに、システムファイル`/etc/services`にリストアップされています。ソケットベースのアプリケーションを作成する場合には、この設定ファイルに記載のないポート番号を選択するように注意する必要があります。

サンプルプログラムではサーバーとクライアントをネットワーク経由で実行しますが、ネットワークソケットの活躍の舞台は、複数のUNIXコンピュータが接続されたネットワークだけとは限りません。インターネットに接続されている1台のマシン（ダイヤルアップ接続でもかまいません）でも、ネットワークソケットを使ってほかのコンピュータと通信することができます。また、ほとんどのUNIXコンピュータではループバックネットワークを使用できるので、ほかのマシンと接続されていない単独のマシン上でもネットワークベースのプログラムを利用することができます。この場合、たとえスタンドアロンであっても、そのUNIXコンピュータは1台だけのコンピュータネットワークで実行されていると考えることができます。ここでは、このループバックネットワークを使ってソケットの動作を検証します。

ループバックネットワークは、一般に`localhost`と呼ばれる1台のコンピュータから構成されます。標準的なIPアドレスの値は127.0.0.1で、これはローカルマシンを意味します。このアドレスは、ネットワーク上のホストの名前とアドレスを記述したネットワークホストファイル`/etc/hosts`に記載されています。

コンピュータの通信先のネットワークには、ハードウェアインタフェースが関連付けられています。1台のコンピュータであっても、それぞれのネットワーク上では異なるネットワーク名を持つことがあり、どのネットワーク上にあるかによってIPアドレスも異なるのがふつうです。たとえば、筆者のマシン`tilde`は3つのネットワークインタフェースを持っており、IPアドレスも3つ持っています。`/etc/hosts`には次のように記述されています。

```
127.0.0.1       localhost            # Loopback
192.168.1.1     tilde.localnet        # Local, private Ethernet
158.152.X.X     tilde.demon.co.uk     # Modem dial-up
```

最初の行はループバックネットワーク上のIPアドレスで、2行目はイーサネットアダプタ経由で接続されたローカルエリアネットワーク上のIPアドレスです。3行目は、インターネットサービスプロバイダに接続する際のモデムリンク上のIPアドレスです。ネットワークソケットベースのプログラムでは、このうちどのネットワーク上のサーバーとも通信することができます。通信先のネットワークに応じて設計を変更する必要はありません。

次に示すのは、前のクライアントプログラムに修正を加えて、ループバックネットワーク経由でネットワークソケットに接続するようにした**client2.c**です。実は、このプログラムにはハードウェアに依存するバグが1つ含まれているのですが、この点についてはあとで詳しく取り上げます。

例題 ネットワーククライアント

1 ヘッダーファイルをインクルードし、必要な変数を用意します。

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

int main()
{
    int sockfd;
    int len;
    struct sockaddr_in address;
    int result;
    char ch = 'A';
```

2 クライアント用ソケットを作成します。

```
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

3 サーバー側と同じ名前でソケットの名前を指定します。

```
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = inet_addr("127.0.0.1");
    address.sin_port = 9734;
    len = sizeof(address);
```

これ以降の部分は**client1.c**と同じです。上の網かけの部分だけを修正してください。

13

プログラムを実行すると、ローカルマシンのポート9734でサーバーが実行されていないので、エラーになります。

```
$ client2
oops: client2: Connection refused
$
```

解説

このクライアントプログラムでは、ヘッダーファイル**netinet/in.h**で定義されている構造体**sockaddr_in**を使って**AF_INET**アドレスを指定し、IPアドレス127.0.0.1のホスト上のサーバーに接続を試みています。関数**inet_addr**は、テキスト表記のIPアドレスをソケットのアドレス指定に適した形式に変換します。アドレス変換関数は、**inet_addr**のほかにもさまざまなものがあります。どのようなものがあるか知りたい場合には、マニュアルページの**inet**を参照してください。

さて、サーバープログラムのほうも、選択したポート番号で接続を待つように変更する必要があります。次に示すのは、必要な修正を加えた**server2.c**です。

例題 ネットワークサーバー

1 ヘッダーファイルをインクルードし、必要な変数を用意します。

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

int main()
{
    int server_sockfd, client_sockfd;
    int server_len, client_len;
    struct sockaddr_in server_address;
    struct sockaddr_in client_address;
```

2 サーバー用に名前のないソケットを作成します。

```
    server_sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

3 **bind**を呼び出してソケットに名前を付けます。

```
server_address.sin_family = AF_INET;
server_address.sin_addr.s_addr = inet_addr("127.0.0.1");
server_address.sin_port = 9734;
server_len = sizeof(server_address);
bind(server_sockfd, (struct sockaddr *)&server_address, server_len);
```

サーバープログラムも、これ以降の部分は**server1.c**と同じです。

解説

このサーバープログラムは、**AF_INET**ドメインソケットを作成してソケットへの接続を待ちます。ソケットは、選択したポート（9734）にバインドされています。接続できるコンピュータは、指定したアドレスによって決まります。ここでは、クライアントプログラムと同様にループバックアドレスを指定しており、ローカルマシンとの通信だけに制限しています。

サーバーがリモートクライアントと通信できるようにするには、接続を許可するIPアドレスの集合を指定する必要があります。特別な値**INADDR_ANY**を使えば、コンピュータが持つすべてのインタフェースを経由した接続を受け入れることができます。もちろん、必要に応じてネットワークインタフェース（内部LANと外部WANなど）を区別することもできます。**INADDR_ANY**は32ビット整数で、アドレス構造体の**sin_addr.s_addr**フィールドで使用することができます。実際に**INADDR_ANY**を使う前に、**client2.c**に含まれているハードウェアに依存するバグについて説明しておきましょう。

13.4.1 ホストバイトオーダーとネットワークバイトオーダー

筆者のコンピュータはIntelプロセッサベースのLinuxマシンです。このマシンで上のサーバープログラムとクライアントプログラムを実行し、**netstat**コマンドを使ってネットワーク接続を確認してみました。コマンドの出力からは、クライアントとサーバーの接続が（必要な処理を終えたあと）クローズされるのを待っているようすがわかります。**netstat**コマンドは、ネットワークを利用できるほとんどのUNIXシステムにインストールされています。

```
$ server2 &
[4] 1225
$ server waiting
client2
server waiting
char from server = B
$ netstat
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        1      0 localhost:1574          localhost:1174          TIME_WAIT
```

注意

> 先へ進む前に、サンプルのサーバープログラムを確実に終了させておいてください。もし実行されたままになっていると、以後のサンプルプログラムでクライアントから接続した際に競合が発生し、期待どおりの結果が得られないことがあります。サーバープロセスをkillするには、次のようにしてハングアップシグナルを送ります。
>
> ```
> kill -1 <process_id>
> ```

netstatコマンドの出力から、サーバーとクライアントの接続に割り当てられているポート番号を知ることができます。**Local Address**の列にはサーバーが、**Foreign Address**の列にはリモートクライアントが表示されています（クライアントは同じマシン上にありますが、ネットワークを介して接続されています）。また、すべてのソケットを区別できるようにするために、一般にクライアントのポートはサーバーのリスンソケットとは異なっており、コンピュータ上で一意のポートが割り当てられます。

さて、**netstat**の出力をよく見てみると、ローカルアドレス（サーバーソケット）は1574になっており、プログラムで指定した9734は使われていません。なぜでしょうか。これは、ポート番号とアドレスがソケットを介して2進数としてやりとりされることが原因です。整数を表すときのバイトオーダーは、コンピュータの種類によって異なります。たとえば、Intelプロセッサが32ビット整数を4つの連続するバイトとしてメモリに格納する際の順序を“1234”だとすると、Motorolaプロセッサでは同じ整数を“4321”という逆の順序でメモリに格納します。整数が格納されているメモリを単純にバイト単位でコピーすれば、異なる種類のコンピュータ間では結果的に値が異なってしまいます。

さまざまな種類のコンピュータが、複数バイトからなる値をネットワーク経由で正しくやりとりするためには、**ネットワークオーダー**を定義する必要があります。そして、クライアントプログラムとサーバープログラムでは、それぞれの内部的な整数表現をネットワークオーダーに変換してから転送を行う必要があります。**netinet/in.h**では、こうした変換を行うための関数がいくつか定義されています。

```
#include <netinet/in.h>

unsigned long int htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int hostshort);
unsigned long int ntohl(unsigned long int netlong);
unsigned short int ntohs(unsigned short int netshort);
```

これらの関数は、16ビット整数と32ビット整数を対象に、ネイティブホスト形式から標準ネットワークオーダーへの変換、およびその逆の変換を行います。関数の名前は、変換の対象と向きを表しています。たとえば、**htonl**は“host to network, long”の略、**htons**は“host to network, short”の略です。ネイティブオーダーとネットワークオーダーが同じコンピュータの場合、これらの関数ではなにも処理が加えられないことになります。

サンプルプログラムのサーバーとクライアントでも、これらの関数を使って16ビットのポート番号のバイトオーダーを正しく扱うようにする必要があります。具体的には、**server2.c**を次のように修正します(さらに、どのIPアドレスからも接続できるように**INADDR_ANY**を使います)。修正後のプログラムは**server3.c**としてください。

```
server_address.sin_addr.s_addr = htonl(INADDR_ANY);
server_address.sin_port = htons(9734);
```

ここでは**INADDR_ANY**を使うようにしましたが、もとの**inet_addr("127.0.0.1")**のままにしておく場合には変換は必要ありません。**inet_addr**はネットワークバイトオーダーで結果を返すからです。クライアントプログラムのほうも次のように修正し、修正後のファイル名を**client3.c**とします。

```
address.sin_port = htons(9734);
```

修正後の**server3**と**client3**を実行すると、今度は正しいポート番号が使われていることがわかります。

```
$ netstat
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        1      0 localhost:9734          localhost:4232          TIME_WAIT
```

ネイティブバイトオーダーとネットワークバイトオーダーが同じコンピュータを使っている場合には、前のサンプルプログラムと結果的に違いはありません。しかし、このような場合でも必ず変換関数を使って、アーキテクチャの異なるクライアントとサーバーの間で正しく処理を行えるようにすることが大切です。

13.5 ネットワーク情報

これまでに示したサンプルプログラムではアドレスとポート番号をハードコーディングしていましたが、汎用的なサーバーとクライアントプログラムではネットワーク情報関数を使って、使用するアドレスとポートを調べることができます。

適切な権限さえあれば、サンプルプログラムのサーバーも、**/etc/services**に記載されている既知のサービスのリストに追加することができます。**/etc/services**にエントリを追加すると、ポート番号に名前を割り当てることができるので、クライアントはポート番号の代わりにサービス名を指定してサーバーに接続できるようになります。

同様に、アドレスを名前に解決するホストデータベース関数を使うと、コンピュータ名からIPアドレスを知ることができます。ホストデータベース関数は、**/etc/hosts**などのネットワーク構成ファイルや、NIS（Network Information Services、以前はYellow Pagesと呼ばれていた）、DNS（Domain Name Service）といったネットワーク情報サービスを参照して必要な処理を行います。

ホストデータベース関数は、インタフェースヘッダーファイル**netdb.h**で次のように宣言されています。

```
#include <netdb.h>

struct hostent *gethostbyaddr(const void *addr, size_t len, int type);
struct hostent *gethostbyname(const char *name);
```

これらの関数が返す構造体には、少なくとも次のメンバがあります。

```
struct hostent {
    char *h_name;          /* name of the host */
    char **h_aliases;      /* list of aliases (nicknames) */
    int h_addrtype;        /* address type */
    int h_length;          /* length in bytes of the address */
    char **h_addr_list     /* list of address (network order) */
};
```

指定されたホストまたはアドレスに対応するデータベースエントリがない場合、ホストデータベース関数はヌルポインタを返します。

サービスとポート番号に関する情報は、次の関数で取得することができます。

```
#include <netdb.h>

struct servent *getservbyname(const char *name, const char *proto);
struct servent *getservbyport(int port, const char *proto);
```

パラメータ**proto**には、サービスに接続するときに使うプロトコルを指定します。具体的には、**SOCK_STREAM** TCP接続の場合には**"tcp"**を、**SOCK_DGRAM** UDPデータグラムの場合には**"udp"**を指定します。

構造体**servent**には、少なくとも次のメンバがあります。

```
struct servent {
    char *s_name;          /* name of the service */
    char **s_aliases;      /* list of aliases (alternative names) */
    int s_port;            /* The IP port number */
    char *s_proto;         /* The service type, usually "tcp" or "udp" */
};
```

実際に、あるコンピュータに関するホストデータベース情報を得るには、**gethostbyname**を呼び出して結果を表示します。このとき、アドレスリストを適切なアドレスのタイプにキャストし、さらにネットワークオーダーで表現された値から印字可能な文字列への変換を行う必要があります。この変換には、次の**inet_ntoa**関数を使います。

```
#include <arpa/inet.h>

char *inet_ntoa(struct in_addr in)
```

inet_ntoa関数は、インターネットホストアドレスを10進数ドット形式の文字列に変換します。エラーが発生した場合には-1を返しますが、POSIXで定義されているエラーはありません。もうひとつ関数を見ておきましょう。

```
#include <unistd.h>

int gethostname(char *name, int namelength);
```

gethostname関数は、**name**で示される文字列に現在のホストの名前を書き込みます。ホスト名はヌルで終わる文字列として返されます。パラメータ**namelength**は文字列**name**の長さを示し、この長さに収まらないホスト名は切り捨てられます。**gethostname**は成功すると0を返し、エラーが発生した場合には-1を返します。**inet_ntoa**同様、**gethostname**の場合にもPOSIXで定義されているエラーはありません。

次のサンプルプログラム**getname.c**は、ホストコンピュータに関する情報を取得します。

例題 ネットワーク情報

1 ヘッダーファイルをインクルードし、必要な変数を用意します。

```
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <netdb.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    char *host, **names, **addrs;
    struct hostent *hostinfo;
```

2 **getname**の起動時に引数が指定されていれば、これをホストに設定し、そうでなければユーザーのマシンをホストに設定します。

13

```
    if(argc == 1) {
        char myname[256];
        gethostname(myname, 255);
        host = myname;
    }
    else
        host = argv[1];
```

3 gethostbynameを呼び出します。必要な情報が見つからなかった場合にはエラーメッセージを表示します。

```
    hostinfo = gethostbyname(host);
    if(!hostinfo) {
        fprintf(stderr, "cannot get info for host: %s\n", host);
        exit(1);
    }
```

4 ホスト名を表示し、ホストが別名を持っている場合にはすべての別名を表示します。

```
    printf("results for host %s:\n", host);
    printf("Name: %s\n", hostinfo -> h_name);
    printf("Aliases:");
    names = hostinfo -> h_aliases;
    while(*names) {
        printf(" %s", *names);
        names++;
    }
    printf("\n");
```

5 該当するホストがIPホストではない場合には、エラーメッセージを表示して終了します。

```
    if(hostinfo -> h_addrtype != AF_INET) {
        fprintf(stderr, "not an IP host!\n");
        exit(1);
    }
```

6 それ以外の場合にはIPアドレスを表示します。

```
    addrs = hostinfo -> h_addr_list;
    while(*addrs) {
        printf(" %s", inet_ntoa(*(struct in_addr *)*addrs));
        addrs++;
    }
    printf("\n");
    exit(0);
}
```

このプログラムでは**gethostbyname**を使っていますが、**gethostbyaddr**を使えば、指定したIPアドレスを持つホストを調べることができます。たとえば、サーバーで**gethostbyaddr**を使うと、クライアントがどこから接続しているかを知ることができます。

解説

このサンプルプログラムは、**gethostbyname**を呼び出してホストデータベースからホスト情報を取得し、ホスト名、別名、およびホストがネットワークインタフェースに割り当てているIPアドレスを表示します。引数に**tilde**を指定して**getname**を実行すると、イーサネットとモデムの2つのインタフェースを持っていることがわかります。

```
$ getname tilde
results for host tilde:
Name: tilde.demon.co.uk
Aliases: tilde
192.9.200.4 158.152.38.110
```

ホスト名に**localhost**を指定すると、ループバックネットワークに関する情報が表示されます。

```
$ getname localhost
results for host localhost:
Name: localhost
Aliases:
127.0.0.1
```

このサンプルプログラムを応用してクライアントプログラムを修正すれば、任意のホストに接続できるようになります。ポート番号を取得する方法もあわせて示すために、これまでのサンプルサーバープログラムではなく、標準サービスに接続するようにクライアントを変更してみましょう。

ほとんどのUNIXシステムでは、システムの日付と時刻を標準サービス**daytime**として利用できるようになっています。クライアントからこのサービスに接続すると、サーバーの現在の日付と時刻を取得することができます。実際にプログラムを作って確かめてみましょう。プログラムの名前は**getdate.c**とします。

13

例題 標準サービスへの接続

1 ヘッダーファイルをインクルードし、必要な変数を用意します。

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <unistd.h>
```

```
int main(int argc, char *argv[])
{
    char *host;
    int sockfd;
    int len, result;
    struct sockaddr_in address;
    struct hostent *hostinfo;
    struct servent *servinfo;
    char buffer[128];

    if(argc == 1)
        host = "localhost";
    else
        host = argv[1];
```

2 ホストアドレスを取得します。見つからなかった場合には、エラーメッセージを表示して終了します。

```
    hostinfo = gethostbyname(host);
    if(!hostinfo) {
        fprintf(stderr, "no host: %s\n", host);
        exit(1);
    }
```

3 目的のホストにdaytimeサービスが存在することを確かめます。

```
    servinfo = getservbyname("daytime", "tcp");
    if(!servinfo) {
        fprintf(stderr,"no daytime service\n");
        exit(1);
    }
    printf("daytime port is %d\n", ntohs(servinfo -> s_port));
```

4 ソケットを作成します。

```
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

5 接続するためのアドレスを用意します。

```
    address.sin_family = AF_INET;
    address.sin_port = servinfo -> s_port;
    address.sin_addr = *(struct in_addr *)*hostinfo -> h_addr_list;
    len = sizeof(address);
```

6 サーバーに接続し、目的の情報を取得します。

```
    result = connect(sockfd, (struct sockaddr *)&address, len);
    if(result == -1) {
        perror("oops: getdate");
        exit(1);
    }

    result = read(sockfd, buffer, sizeof(buffer));
    printf("read %d bytes: %s\n", result, buffer);

    close(sockfd);
    exit(0);
}
```

このgetdateを使えば、任意のホストの時刻を取得することができます。

```
$ getdate tilde
daytime port is 13
read 26 bytes: Sun Jan 14 13:09:47 1996
$
```

解説

このプログラムでは、接続先のホストを指定することができます。daytimeサービスのポート番号は、ネットワークデータベース関数getservbynameで取得します。getservbynameは、ホスト情報を返すgethostbyname同様、ネットワークサービスに関する情報を返します。getdateは、引数に指定されたホストが持つアドレスのリストのうち、最初のアドレスを使って接続を試みます。成功した場合には、daytimeサービスが返す情報（UNIXの日付と時刻を表す文字列）を読み取ります。

13.5.1 インターネットデーモン

UNIXシステムでは、しばしば1つのスーパーサーバーでさまざまなネットワークサービスを提供しています。**インターネットデーモン**（inetd）と呼ばれるこのスーパーサーバーは、同時に複数のポートアドレスで接続をリスンしており、提供しているサービスのひとつにクライアントが接続すると、適切なサーバーを起動するようになっています。インターネットデーモンを使うと、必要に応じてinetdからサーバーを起動することができるので、常時サーバーを実行しておく必要がありません。次に示すのは、インターネットデーモンから起動するサーバーを記述したinetdの設定ファイルの一部です。

```
#
# <service_name> <sock_type> <proto> <flags> <user> <server_path> <args>
#
daytime   stream  tcp nowait  root    internal
daytime   dgram   udp wait    root    internal
#
# These are standard services.
#
ftp   stream  tcp nowait  root    /usr/sbin/tcpd  wu.ftpd -l -i -a
telnet    stream  tcp     nowait  root    /usr/sbin/tcpd      in.telnetd
#
# End of inetd.conf.
```

この設定ファイルを見ると、サンプルプログラムgetdateで利用したdaytimeサービスは、実際にはinetd自体が処理し（internal）、SOCK_STREAM（tcp）とSOCK_DGRAM（udp）の両方で利用できることがわかります。

一方、ファイル転送プロトコルサービスftpは、SOCK_STREAMソケット経由でのみ利用でき、外部プログラムを起動するようになっています。上の設定ファイルの場合、クライアントがftpポートに接続すると、inetdによってwu.ftpdが起動されます。

13.5.2 ソケットのオプション

ソケット接続の動作を制御するためのオプションは非常に数が多く、ここではその一部だけを取り上げます。ソケットオプションを操作するには、次の関数を使います。

```
#include <sys/socket.h>

int setsockopt(int socket, int level, int option_name,
        const void *option_value, size_t option_len);
```

setsockoptでは、プロトコルの階層構造のさまざまなレベルでソケットオプションを設定できます。たとえば、ソケットレベルでオプションを設定するには、パラメータlevelにSOL_SOCKETを設定します。TCP、UDPなどのプロトコルレベルでオプションを設定するには、パラメータlevelにプロトコル番号を設定します。プロトコル番号は、ヘッダーファイルnetinet/in.hで定義されているものを使うか、またはgetprotobyname関数で取得します。

パラメータoption_nameには、設定するオプションを指定します。option_valueはoption_lenバイトの長さを持つ値で、setsockoptでは手を加えられずに、下位のプロトコルハンドラにそのまま渡されます。

ソケットレベルのオプションはsys/socket.hで定義されており、次のようなものがあります。

- SO_DEBUG　　デバッグ情報を有効にする。
- SO_KEEPALIVE　　定期的に転送を行って接続をアクティブに保つ。
- SO_LINGER　　転送を終えてからクローズする。

SO_DEBUGとSO_KEEPALIVEでは、整数値のoption_valueを使ってオプションを有効または無効にします。有効にする場合には1を、無効にする場合には0を指定します。SO_LINGERでは、sys/socket.hで定義されているlinger構造体を使ってオプションの状態と遅延間隔を指定します。

setsockoptは成功すると0を返し、失敗した場合には -1を返します。オプションとエラーの詳しい説明については、マニュアルページを参照してください。

13.6 複数のクライアント

これまでの部分では、ソケットを使ってローカルなクライアントサーバーシステムやネットワークを介したクライアントサーバーシステムを実装する方法を説明しました。いったん確立されたソケット接続のふるまいは、オープンされた低水準ファイルデスクリプタと似ており、双方向のパイプとも多くの共通項を持っています。

ソケットでは、1つのサーバーに複数のクライアントが同時に接続するケースを考慮する必要があります。すでに見たように、サーバーがクライアントから新しい接続を受け入れると、新しくソケットが作成され、もとのリスンソケットはほかのクライアントからの接続用にそのまま残されます。接続がサーバーによってすぐに受け入れられない場合、その接続は保留中としてキューに保持されます。

もとのソケットが利用可能であることと、ソケットがファイルデスクリプタのようにふるまうことを利用すれば、同時に複数のクライアントにサービスを提供することができます。具体的には、サーバー側でforkを呼び出します。サーバーがforkを使って自分自身のコピーを作成すると、オープンされたソケットは新しい子プロセスに受け継がれます。子プロセスは接続してきたクライアントと通信し、サーバーはほかのクライアントからの接続を待ちます。では、実際にサーバープログラムに変更を加えてこのアイデアを実現してみましょう。変更箇所はそれほど多くありません。

子プロセスを作成しても子プロセスの終了を待つわけではないので、サーバー側ではSIGCHLDシグナルを無視し、ゾンビプロセスが発生しないようにする必要があります。作成するプログラムの名前はserver4.cとします。

例題 1つのサーバーと複数のクライアント

1 最初の部分は前のサンプルプログラムとほとんど同じですが、ヘッダーファイル`signal.h`をインクルードします。

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <netinet/in.h>
#include <signal.h>
#include <unistd.h>

int main()
{
    int server_sockfd, client_sockfd;
    int server_len, client_len;
    struct sockaddr_in server_address;
    struct sockaddr_in client_address;

    server_sockfd = socket(AF_INET, SOCK_STREAM, 0);

    server_address.sin_family = AF_INET;
    server_address.sin_addr.s_addr = htonl(INADDR_ANY);
    server_address.sin_port = htons(9734);
    server_len = sizeof(server_address);
    bind(server_sockfd, (struct sockaddr *)&server_address, server_len);
```

2 接続キューを作成し、子プロセスの終了は無視してクライアントの接続を待ちます。

```
    listen(server_sockfd, 5);

    signal(SIGCHLD, SIG_IGN);

    while(1) {
        char ch;

        printf("server waiting\n");
```

3 接続を受け入れます。

```
        client_sockfd = accept(server_sockfd,
            (struct sockaddr *)&client_address, &client_len);
```

4 `fork`を呼び出してクライアント用のプロセスを作成し、親か子かの判定をします。

```
        if(fork() == 0) {
```

5 子プロセスでは、`client_sockfd`を使ってクライアントへの読み書きを行うことができます。5秒間スリープしているのは、実際のサーバーではこの部分でなんらかの必要な処理を行うことになるためです。

```
            read(client_sockfd, &ch, 1);
            sleep(5);
            ch++;
            write(client_sockfd, &ch, 1);
            close(client_sockfd);
            exit(0);
        }
```

6 子プロセスでなければ親です。接続してきたクライアントに対して親が行う処理はありません。

```
        else {
            close(client_sockfd);
        }
    }
}
```

5秒間のスリープは、クライアントの要求に応えてサーバー側で計算やデータベースアクセスを行う場合の遅延を想定したものです。`fork`を使わないサーバープログラムで同じように5秒間スリープしたとすると、`client3`を1つ実行するたびに5秒経過することになります。`fork`を使う上のサーバープログラムでは、複数の`client3`からの接続を同時に処理することができ、処理時間も全体でわずかに5秒を超える程度ですみます。

```
$ server4&
[2] 10694
$ server waiting
client3 & client3 & client3 & ps -ax
[3] 10695
[4] 10696
[5] 10697
server waiting
server waiting
server waiting
  PID TTY STAT  TIME COMMAND
10694  p4 S    0:00 server4
10695  p4 S    0:00 client3
10696  p4 S    0:00 client3
10697  p4 S    0:00 client3
10698  p4 R    0:00 ps -ax
10699  p4 S    0:00 server4
10700  p4 S    0:00 server4
10701  p4 S    0:00 server4
$ char from server = B
char from server = B
```

13

```
char from server = B
ps -ax
  PID TTY STAT  TIME COMMAND
10694  p4 S    0:00 server4
10704  p4 R    0:00 ps -ax
[3]   Done                  client3
[4]-  Done                  client3
[5]+  Done                  client3
$
```

解説

このサーバープログラムでは、各クライアントを処理するために新しく子プロセスを作成するので、サーバーが待機中であることを示すメッセージが複数表示されています。**ps**コマンドの出力（一部省略しています）では、メインの**server4**のプロセス（PIDは10694）が新しいクライアントの接続を待つ一方で、**client3**の3つのプロセスがメインのサーバーの3つの子プロセスによって処理されていることがわかります。5秒間の休止後、すべてのクライアントは結果を取得して終了します。サーバーの子プロセスも終了し、最後にメインのサーバープロセスだけが残ります。

このサンプルプログラムでは、**fork**を使って複数のクライアントを処理しています。**fork**を使う方法は、データベースアプリケーションでは必ずしも最適な解決方法ではないかもしれません。データベースアプリケーションは巨大であることが多く、作成された複数の子プロセスからデータベースにアクセスする際にそれぞれの処理を互いにどのように協調させるかも考慮しなければならないからです。こうした点を考えると、1つのサーバーがクライアントからの要求を待ちながら、ブロックされることなく複数のクライアントを処理できる方法を用意する必要があります。それには、オープンされた複数のファイルデスクリプタを同時に扱う手段が必要になりますが、それを可能にするのが次で説明する**select**です。**select**を使ったテクニックは、ソケットアプリケーション以外にも応用することができます。

13.7 select

UNIXアプリケーションを作成する場合、複数の入力装置の状態を調べ、次に行う処理を決定しなければならないことがあります。たとえば、端末エミュレータなどの通信プログラムの場合には、キーボードとシリアルポートを同時に、かつ効率的に読み取る必要があります。シングルユーザーシステムの場合には、ビジーウェイト状態でループしながら入力データを検査し、データがくればそれを読み取るといった処理を行うことも可能です。しかし、UNIXシステムのようなマルチユーザー／マルチタスクシステムでは、この方法ではCPU時間を無駄に消費してしまいます。

このような場合、**select**システムコールを使うと、同時に複数の低水準ファイルデスクリプタを対象に入力データの到着（または出力の完了）を待つことができます。端末エミュレータプログラムで**select**を使えば、なにか処理すべきことが発生するまでの間、プログラムのブロックが可能になります。同様に、クライアントサーバーシステムでは、オープンされた複数のソケットを対象に要求を待つことによって、1つのサーバーで複数のクライアントを処理できるようになります。

selectシステムコールの操作の対象はデータ構造体**fd_set**で、これはオープンされたファイルデスクリプタの集合を表します。この集合を操作するために、次のようなマクロが定義されています。

```
#include <sys/types.h>
#include <sys/time.h>

void FD_ZERO(fd_set *fdset);
void FD_CLR(int fd, fd_set *fdset);
void FD_SET(int fd, fd_set *fdset);
int FD_ISSET(int fd, fd_set *fdset);
```

名前から想像できるように、**FD_ZERO**は**fdset**を空に初期化します。**FD_SET**と**FD_CLR**は、**fd**で指定されたファイルデスクリプタを**fdset**に追加または削除します。**FD_ISSET**は、**fd**で指定されたファイルデスクリプタが**fdset**で示される**fd_set**の要素であれば、0以外の値を返します。1つの**fd_set**構造体に保持できるファイルデスクリプタの最大数は、定数**FD_SETSIZE**で定義されています。

selectでは、無期限にブロックされるのを防ぐためにタイムアウト値を指定することができます。タイムアウト値の指定には、次に示す**struct timeval**構造体を使います。**struct timeval**構造体は**sys/time.h**で定義されています。

```
struct timeval {
  time_t    tv_sec;     /* seconds */
  long      tv_usec;    /* microseconds */
}
```

time_t型は、**sys/types.h**で整数型として定義されています。

selectシステムコールのプロトタイプは次のとおりです。

```
#include <sys/types.h>
#include <sys/time.h>

int select(int nfds, fd_set *readfds, fd_set *writefds,
fd_set *errorfds, struct timeval *timeout);
```

selectは、指定した集合に含まれる任意のファイルデスクリプタを対象に、読み取りまたは書き込みの準備ができているかどうか、あるいはエラーが発生しているかどうかを調べます。ま

た、オプションの指定によっては、いずれかのファイルデスクリプタの準備が整うまでブロックします。

パラメータ`nfds`には、調べるファイルデスクリプタの数を指定します。実際に調べられるファイルデスクリプタは、0から`nfds-1`までです。パラメータ`nfds`のあとに指定する3つのファイルデスクリプタ集合には、ヌルポインタを指定することができます。ヌルポインタが指定された場合、対応するテストは実行されません。

`select`は、`readfds`に含まれるいずれかのファイルデスクリプタで読み取りの準備ができている場合、`writefds`のいずれかのファイルデスクリプタで書き込みの準備ができている場合、または`errorfds`のいずれかのファイルデスクリプタでエラーが発生している場合に呼び出しから戻ります。これらの条件のいずれにも該当しない場合には、`timeout`で指定された時間が経過したあとに戻ります。パラメータ`timeout`にヌルポインタを指定した場合、ファイルデスクリプタに対して読み取りや書き込みなどの操作がなにも行われなければ、呼び出しは永久にブロックされます。

呼び出しから戻ると、ファイルデスクリプタの集合には修正が加えられ、読み取りや書き込みの準備ができているファイルデスクリプタ、またはエラーの発生しているファイルデスクリプタはどれかを特定することができます。返されたファイルデスクリプタの集合を調べるには、すでに示したマクロ`FD_ISSET`を使います。`timeout`の値を修正することで次のタイムアウトまでの残り時間を指定することもできますが、これはX/Open仕様にはない動作です。タイムアウトが発生した場合、すべてのファイルデスクリプタ集合は空になります。

`select`は、修正後の集合に含まれるデスクリプタの総数を返します。エラーが発生した場合には-1を返し、エラーの内容を示す値が`errno`に設定されます。エラーとしては、**EBADF**（無効なデスクリプタ）、**EINTR**（割り込み）、**EINVAL**（`nfds`または`timeout`の値が不正）が発生する可能性があります。

注意

> Linuxでは、`timeout`によって示される構造体を変更して残り時間を表しますが、ほとんどのUNIXシステムではこの構造体を変更しません。`select`を使っている多くの既存のコードでは、`timeval`構造体をいったん初期化したあと、再初期化を行わずにそのまま使い続けます。このようなコードは、タイムアウトが発生するたびに`timeval`構造体の内容を変更するLinuxでは、正しく動作しない可能性があります。`select`関数を使うコードを記述または移植する場合には、この点に注意する必要があります。なお、どちらの動作もそれ自体間違っているわけではありません（ただ動作が異なるだけです）。

次に示すのは、`select`システムコールを使ったごく簡単なサンプルプログラム`select.c`です。このプログラムでは、タイムアウトを2.5秒に設定してキーボード（ファイルデスクリプタ0、つまり標準入力）を読み取ります。キーボードの読み取りは、入力の準備ができている場合にのみ行います。ほかのデスクリプタも対象に加えれば、シリアル回線やパイプ、ソケットといったものにも応用することができます。

例題 select

1 ヘッダーファイルをインクルードし、必要な変数を用意したあと、**inputs**を初期化してキーボードを集合に加えます。

```
#include <sys/types.h>
#include <sys/time.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <unistd.h>

int main()
{
    char buffer[128];
    int result, nread;

    fd_set inputs, testfds;
    struct timeval timeout;

    FD_ZERO(&inputs);
    FD_SET(0,&inputs);
```

2 標準入力からの入力を最大2.5秒間待ちます。

```
    while(1) {
        testfds = inputs;
        timeout.tv_sec = 2;
        timeout.tv_usec = 500000;

        result = select(FD_SETSIZE, &testfds, (fd_set *)0, (fd_set *)0, &timeout);
```

3 2.5秒が経過したら、結果をテストします。入力がない場合には、ループを繰り返し実行します。エラーが発生していた場合には、プログラムを終了します。

```
        switch(result) {
        case 0:
            printf("timeout\n");
            break;
        case -1:
            perror("select");
            exit(1);
```

4 待っている間にファイルデスクリプタに対してなんらかの操作が行われた場合には、**stdin**から入力を読み取り、行末文字を受け取るたびに画面にエコーします。入力が**Ctrl - D**だった場合には、プログラムを終了します。

```
        default:
            if(FD_ISSET(0,&testfds)) {
                ioctl(0,FIONREAD,&nread);
                if(nread == 0) {
                    printf("keyboard done\n");
                    exit(0);
                }
                nread = read(0,buffer,nread);
                buffer[nread] = 0;
                printf("read %d from keyboard: %s", nread, buffer);
            }
            break;
        }
    }
}
```

プログラムを実行すると、2.5秒おきに`timeout`というメッセージが表示されます。キーボードからなにか入力すると、プログラムが標準入力を読み取り、入力された文字を表示します。ほとんどのシェルは、ユーザーが**Return**キーを押した時点で入力をプログラムに渡すので、上のプログラムでも**Return**キーを押すと、入力した内容が表示されます。**Return**キー自体もほかの文字と同様に読み取られて処理されます。

```
$ select
timeout
hello
read 6 from keyboard: hello
fred
read 5 from keyboard: fred
timeout
^D
keyboard done
$
```

解説

このプログラムでは、`select`を呼び出して標準入力の状態を調べています。タイムアウト値は2.5秒に設定されており、この間隔で画面に`timeout`というメッセージが表示されます。メッセージが表示されるときは、`select`が0を返しています。ファイルの終わりに達した場合、標準入力のファイルデスクリプタでは入力の準備はできていますが、読み取るべき文字はありません。

13.7.1 selectを使った複数のクライアントへの対処

サンプルサーバープログラムでselectを使えば、子プロセスを作成しなくても複数のクライアントを同時に処理することができます。

具体的には、リスンソケットとクライアントの接続ソケットの両方に対して同時にselectを使用します。なんらかの変化があった場合には、FD_ISSETを使って対象となるファイルデスクリプタをすべて調べれば、該当するファイルデスクリプタを特定することができます。

リスンソケットで入力の準備ができている場合には、クライアントが接続を試みていることを意味します。この場合は、ブロックを心配せずにacceptを呼び出すことができます。クライアントのデスクリプタで入力の準備ができている場合には、クライアントの要求が保留中であることを意味します。この場合は、要求を読み取って処理することができます。読み取るべきバイト数が0の場合には、クライアントプロセスが終了したことを意味します。この場合は、ソケットをクローズしてデスクリプタの集合から削除することができます。

次に実際のコードを示します。プログラムの名前はserver5.cとします。

例題 selectを使った複数のクライアントへの対処

1 signal.hの代わりにsys/time.hとsys/ioctl.hをインクルードし、selectで使う変数の宣言を追加します。

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <netinet/in.h>
#include <sys/time.h>
#include <sys/ioctl.h>
#include <unistd.h>

int main()
{
    int server_sockfd, client_sockfd;
    int server_len, client_len;
    struct sockaddr_in server_address;
    struct sockaddr_in client_address;
    int result;
    fd_set readfds, testfds;
```

2 サーバー用ソケットを作成し、名前を付けます。

```
    server_sockfd = socket(AF_INET, SOCK_STREAM, 0);

    server_address.sin_family = AF_INET;
    server_address.sin_addr.s_addr = htonl(INADDR_ANY);
    server_address.sin_port = htons(9734);
```

```
server_len = sizeof(server_address);

bind(server_sockfd, (struct sockaddr *)&server_address, server_len);
```

3 接続キューを作成し、readfdsを初期化してserver_sockfdをデスクリプタの集合に加えます。

```
listen(server_sockfd, 5);

FD_ZERO(&readfds);
FD_SET(server_sockfd, &readfds);
```

4 クライアントおよび要求を待ちます。パラメータtimeoutにはヌルポインタを指定しているので、タイムアウトは発生しません。selectが1より小さい値を返した場合には、エラーメッセージを表示してプログラムを終了します。

```
while(1) {
    char ch;
    int fd;
    int nread;

    testfds = readfds;

    printf("server waiting\n");
    result = select(FD_SETSIZE, &testfds, (fd_set *)0,
        (fd_set *)0, (struct timeval *) 0);

    if(result < 1) {
        perror("server5");
        exit(1);
    }
```

5 なんらかの変化があった場合には、対象となるデスクリプタに対して順にFD_ISSETを適用し、該当するデスクリプタを特定します。

```
    for(fd = 0; fd < FD_SETSIZE; fd++) {
        if(FD_ISSET(fd,&testfds)) {
```

6 該当するデスクリプタがserver_sockfdの場合には、新しい接続の要求なので、acceptを呼び出し、返されたclient_sockfdをデスクリプタの集合に追加します。

```
            if(fd == server_sockfd) {
                client_sockfd = accept(server_sockfd,
                    (struct sockaddr *)&client_address, &client_len);
                FD_SET(client_sockfd, &readfds);
                printf("adding client on fd %d\n", client_sockfd);
            }
```

7 該当するデスクリプタが`server_sockfd`でなければ、クライアントからの要求です。読み取るべきデータがない場合には、クライアントが終了しているので、該当するデスクリプタを集合から削除します。それ以外の場合には、クライアントにサービスを提供します。

```
                else {
                    ioctl(fd, FIONREAD, &nread);

                    if(nread == 0) {
                        close(fd);
                        FD_CLR(fd, &readfds);
                        printf("removing client on fd %d\n", fd);
                    }

                    else {
                        read(fd, &ch, 1);
                        sleep(5);
                        printf("serving client on fd %d\n", fd);
                        ch++;
                        write(fd, &ch, 1);
                    }
                }
            }
        }
    }
}
```

注意

> サンプルプログラムではループを使って合計1000ほどのデスクリプタを調べていますが、この中には接続されていないデスクリプタも含まれており、こうしたものも含めてすべてのデスクリプタに対してテストを実行するのは時間の無駄です。実際にプログラムを作成する場合には、接続中の`fd`の最大値を保持する変数を用意したほうがよいでしょう。

プログラムを実行すると、1つのプロセスで複数のクライアントが順次処理されていることがわかります。

```
$ server5 &
[1] 15267
$ server waiting
client3 & client3 & client3 & ps -ax
[2] 15268
[3] 15269
[4] 15270
adding client on fd 5
server waiting
  PID TTY STAT  TIME COMMAND
15267  p4 S    0:00 server5
15268  p4 S    0:00 client3
```

```
15269  p4 S    0:00 client3
15270  p4 S    0:00 client3
15271  p4 R    0:00 ps -ax
$ serving client on fd 5
server waiting
adding client on fd 6
server waiting
adding client on fd 7
char from server = B
serving client on fd 6
server waiting
removing client on fd 5
char from server = B
serving client on fd 7
server waiting
removing client on fd 6
server waiting
char from server = B
removing client on fd 7
server waiting

[2]   Done                    client3
[3]-  Done                    client3
[4]+  Done                    client3
$
```

13.8 この章のまとめ

この章の冒頭で取り上げた電話の呼び出しの例を使って、ソケット接続について簡単にまとめておきましょう。

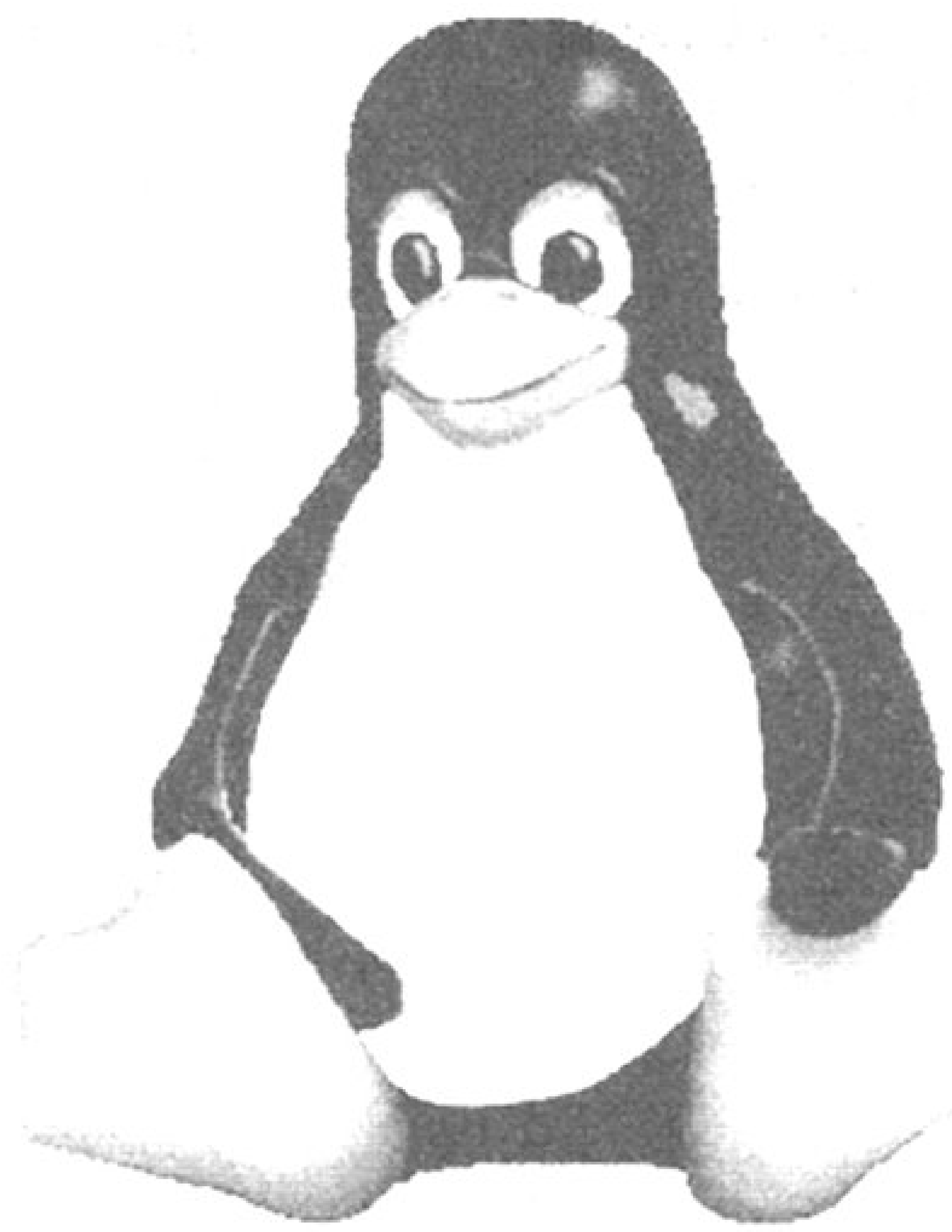
表13.1 電話の呼び出しとソケット接続の対応

電話	ネットワークソケット
会社への電話の着信	IPアドレス127.0.0.1に接続
受付の対応	**inetd**によって接続が確立される
通話先として経理部を指定	指定ポート(9734)に振り向けられる
経理部長の応答	サーバーが**select**から戻る
該当する担当者の内線に転送	サーバーが**accept**を呼び出して新しいソケットを作成
受付と経理部長は新しい着信を待機	コンピュータとサーバープロセスは新しいクライアントからの接続を待機

この章では、System V IPCとは異なるプロセス間通信の方法として、ソケットについて説明しました。ソケットを使うと、ネットワーク経由で実行可能な本当の意味での分散クライアントサーバーアプリケーションを開発することができます。また、ホストデータベース情報関数も取

り上げ、インターネットデーモンによるUNIX標準システムサービスの提供のしくみについて説明しました。例として用いたクライアントサーバープログラムでは、ネットワークや複数のクライアントの扱い方を具体的に示しました。

最後に`select`システムコールを取り上げ、オープンされた複数のファイルデスクリプタやソケットの集合を対象として、入出力の準備が整っているかどうかを調べる方法を示しました。



付録A

Xプログラミングの概要

X Window System（または**X**）は、UNIXで広く使われているグラフィカル環境です。Xは、今日のほとんどのUNIXシステムやLinuxディストリビューションに付属しています。

残念ながら本書では、Xプログラミングについて詳しく取り上げることはできません。そこでこの付録では、読者が今後Xプログラミングに取り組もうとする際の指針となるように、XとXプログラミングの概要について簡単に触れておきます。ここでは、すでに読者がXを使っていることを前提として、プログラマの観点から見たXについて説明します。取り上げる項目は次のとおりです。

- Xのコンセプト
- Xのプログラミングモデル

A.1 Xとは

Xは、グラフィカルなプログラムに対する統一環境を提供することを目的として、MITで作成されました。これまでにMicrosoft Windowsが動作しているPCやApple Macintoshコンピュータを使った経験があれば、グラフィカルユーザーインタフェース（GUI）の考え方にはもう慣れているでしょう。実際、Windowsユーザーならば、それほど苦労することなくMacのインタフェースを使うことができます。しかし、プログラマから見た場合、事情はまったく異なります。

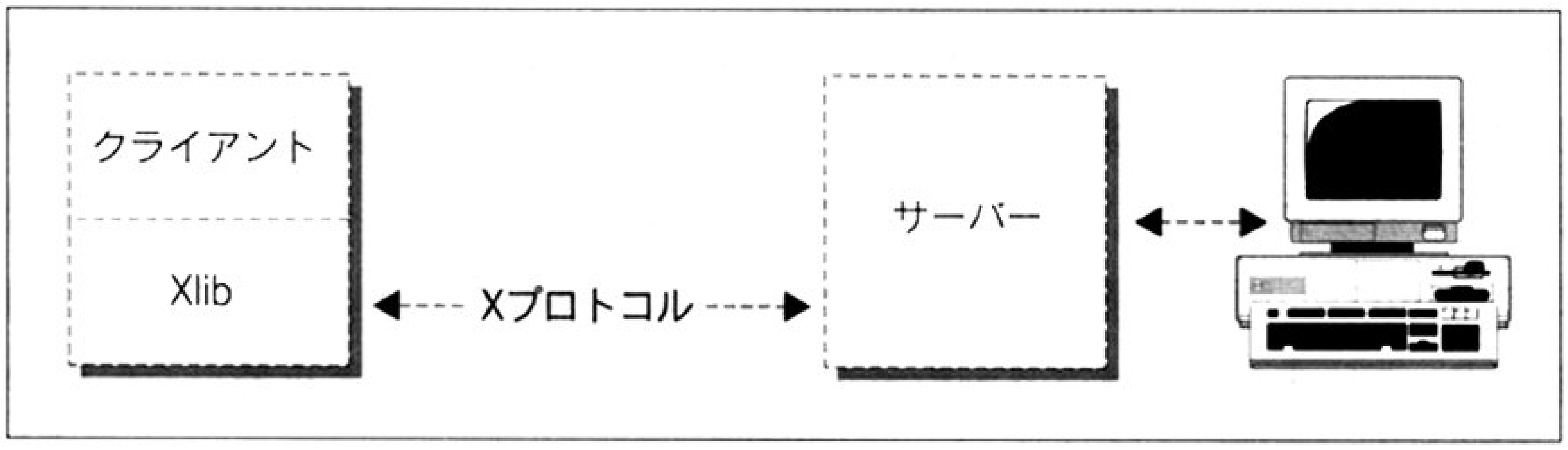
各システムのウィンドウ環境は、それぞれ独自の方法でプログラムされています。ディスプレイの扱いや、プログラムがユーザーと対話する方法はシステムごとに異なります。もちろんどのシステムでも、画面上にウィンドウを開いたり、ウィンドウを操作したりするための機能は用意されています。しかし、そのために使われる関数は異なります。このため、適切なツールが利用できる場合は別として、複数のウィンドウシステムで動作するアプリケーションを記述することは現実的には非常に困難です。

X Window Systemは、メインフレームやミニコン、ワークステーション上の独自インタフェースシステムに起因するさまざまな問題を克服するために、広く一般に公開され、今では多くのシステムに実装されています。X Window Systemでは、クライアントサーバーモデルに基づいたプログラミングスタイルを採用しており、ハードウェアに依存するコンポーネントとアプリケーションプログラムとを明確に分離しています。

X Window Systemを構成する主な要素は、次の4つです。

- Xサーバー
- Xクライアント
- Xプロトコル
- Xライブラリ

以下では、それぞれの要素について簡単に説明します。



図A.1　Xのコンセプト

A.1.1 Xサーバー

Xサーバー、または**Xディスプレイサーバー**は、アプリケーションユーザーのコンピュータ上で実行されるプログラムで、グラフィカルディスプレイハードウェアの制御、および入出力の部分を担当します。Xサーバーは、Xクライアントアプリケーションからの要求に応えて画面への描画を行い、キーボードやマウスからの入力を読み取ります。入力やマウスの移動、ボタンの押し下げといったユーザーの操作は、クライアントプログラムに伝えられます。

通常、Xサーバーは、ハードウェア構成ごとに異なるものを使います。LinuxなどのPCベースのシステムで最も一般的なXの実装は、**XFree86**（`http://www.xfree86.org/`）です。XFree86は、PCで使われているさまざまなビデオカード用のXサーバーを数多く提供しています。たとえば、S3ベースのカード用には、`XF86_S3`というXサーバーがあります。こうしたさまざまなXサーバーを利用できるLinuxユーザーは、XFree86の開発者たちに感謝しなければなりません。

A.1.2 Xプロトコル

Xクライアントアプリケーションと Xディスプレイサーバーの間のすべてのやりとりは、メッセージの交換によって実現されています。**Xプロトコル**は、これらのメッセージの種類やその使い方を定めたものです。Xプロトコルは、同一マシン上のクライアントとサーバーの間だけでなく、ネットワークを介して利用することができ、この点はX Window Systemのたいへん優れた特徴のひとつとなっています。たとえば、古い非力なパーソナルコンピュータやX端末（Xサーバー実行用の専用マシン）を使っている場合でも、ネットワーク上のより強力なコンピュータでXクライアントプログラムを実行し、自分のローカルマシンではプログラムに対する指示とプログラムからの出力の表示だけを行うことができます。

A.1.3 Xlib

Xプロトコルは、Xサーバーを実装する開発者にとっては重要な意味を持っていますが、ほとんどのXアプリケーションは、プログラミングインタフェースとしてCの関数ライブラリを使っています。これが**Xlib**です。Xlibは、Xプロトコルによるメッセージ交換を行うためのAPIを提供します。Xlibは、画面に描画したりマウス操作に応答したりするだけで、それ自体ではたいした機能を持っていません。メニューやボタン、スクロールバーなどを使いたい場合には、自分でコードを記述する必要があります。

逆にいえば、Xlibは特定のスタイルのGUIを強制していません。Xlibは、プログラマが選択したスタイルを作成するための媒体として機能します。

A.1.4 Xクライアント

Xクライアントは、コンピュータの表示リソースと入力リソースを使うアプリケーションプログラムのことです。この場合のコンピュータは、Xクライアントが実行されているコンピュータであってもそうでなくてもかまいません。Xクライアントは、表示リソースと入力リソースを管理しているXサーバーに対してこれらのリソースへのアクセスを要求します。サーバーでは、同時に複数のクライアントから送られた要求を処理することがあるので、キーボードとマウスの使用に関してクライアント間で調停を行わなければなりません。クライアントプログラムは、Xプロトコルメッセージを使ってサーバーと通信します。これらのメッセージはXlibの関数を使って送受信されます。

A.1.5 Xツールキット

手軽にXプログラムを作成するための手段として見た場合、Xlibプログラミングインタフェースはあまり優れたツールとはいえません。Xlibが提供するのはMicrosoft Windows SDKのような低レベルインタフェースであり、複雑なプログラムを作成しても実現できることはわずかです。筆者が持っているある本には、Xlibを使って“Hello World”プログラムを作成する例がのっていますが、ウィンドウに`Hello World`というテキストと`Exit`ボタンを表示するだけで、コードの量は5ページにものぼっています。

Xlibを使ってプログラムを作成したことのあるプログラマなら、必ずやほかになにかよい方法はないだろうかと思うに違いありません。もちろん、より手軽にプログラムを作成する方法はあります。ボタン、スクロールバー、メニューといったよく使うユーザーインタフェース要素の実装は、すでに何度も試みられています。こうしたインタフェース要素（**ウィジェット**）のコレクションは、一般に**Xツールキット**と呼ばれています。その中で最もよく知られているものとして、Xに付属するXt Intrinsicsと、2つの商業ベースの製品、SunのOpenLookとOSF/Motifがあります。

Xtはフリーのライブラリで、アプリケーションプログラミングを容易にするための中間層としての役割を果たします。

OpenLookはSunが提供するツールキットで、独自の外観と操作性を持っています。OpenLookは、Xtに似たXviewというライブラリを土台に構築されています。

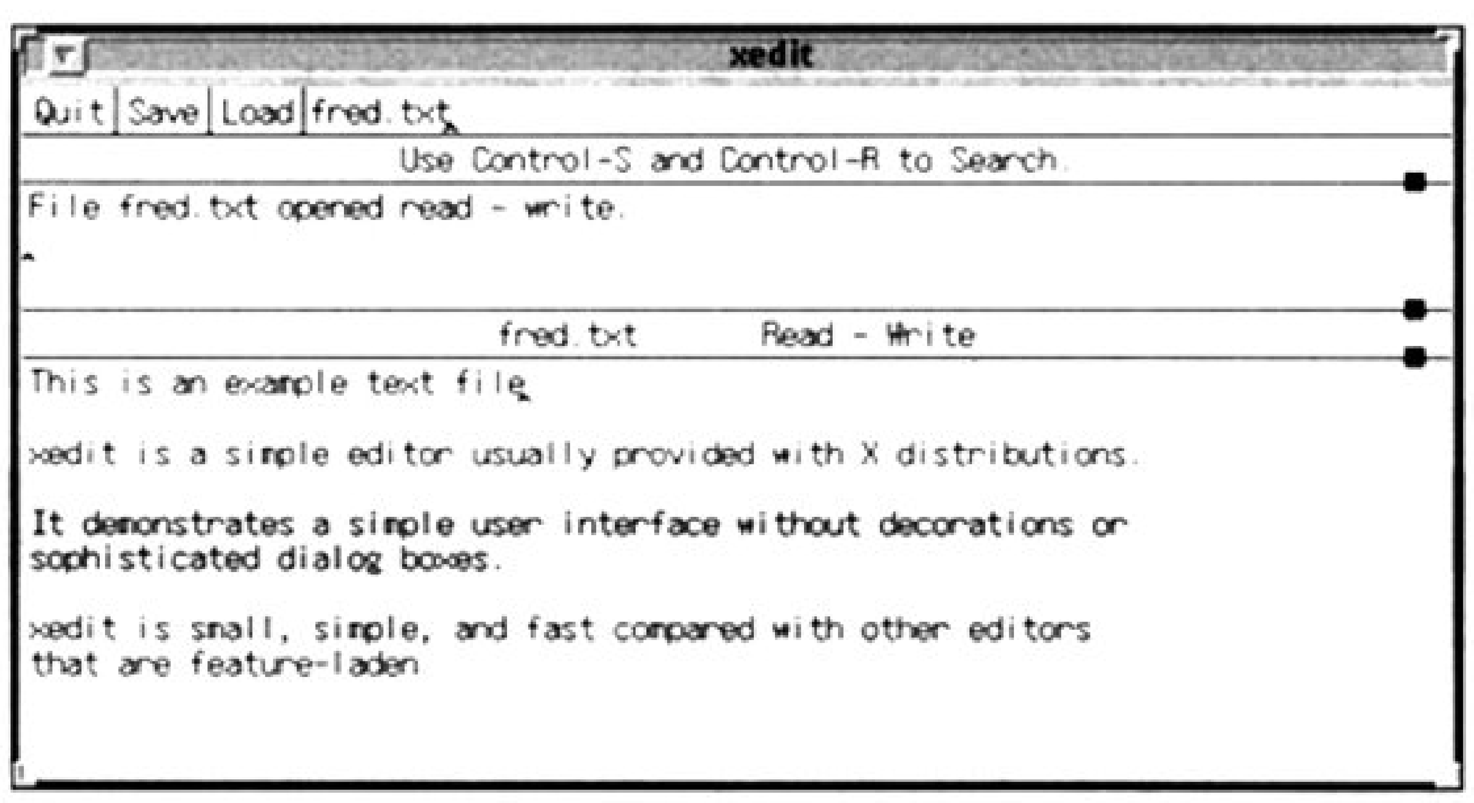
MotifはUNIXデスクトップに共通の外観と操作性を与えるために設計されたOSFの規格で、Xtの上に構築されています。Motifには2つの主要コンポーネントがあります。ひとつは、Xt関数で使われる定数を定義した一連のインクルードファイルです。もうひとつは、ダイアログやメニューなどのインタフェース要素を簡単に作成するための関数ライブラリです。Motifではプログラミングスタイルも定義しています。このプログラミングスタイルは、実際にMotifを使うかどうかにかかわらず、すべてのプログラマが参考にできるものです。

メモ

ほかの商用UNIXシステムベンダー同様、現在ではSunもMotifを使ったCDE（Common Desktop Environment）を採用しています。このため、OpenLookはその役割を終え、徐々に消えていくと考えられます。OpenLookの一部はフリーでの利用が可能になっており、Linuxのディストリビューションにも含まれています。

各Xツールキットでは、独自の外観と操作性を持つ一連のウィジェットを実装しています。画面要素の実装としては、Xtのようにフラットでプレーンなものや、Motifのように影のついた立体的なものがあります。

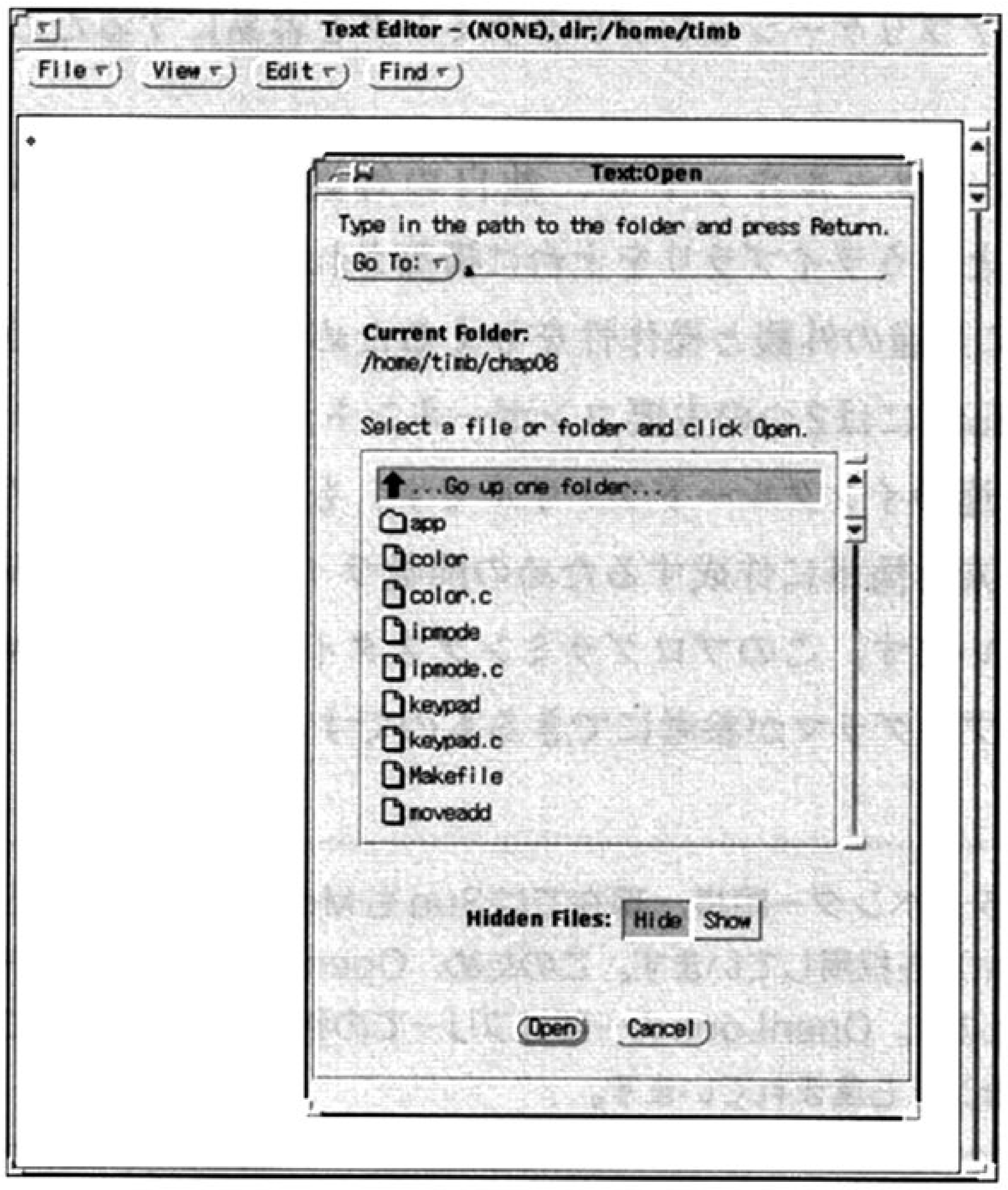
ツールキットによる印象の違いは、Linuxで利用できる2つのテキストエディタ、**`xedit`**と**`textedit`**の画面を見るとよくわかります。**`xedit`**は素朴なユーザーインタフェースを持つシンプルなエディタです。ファイルをロードするには、ボックスにファイル名を入力し、**`Load`**ボタンを押す必要があります。



図A.2　xeditの画面

一方、**`textedit`**はSunのOpenWindowsで提供されているエディタで、OpenLookツールキットを使っています。**`textedit`**にはファイルを開くためのダイアログボックスもあります。このダイアログ ボックスを使うと、表示されたファイルの中から目的のファイルを選択することができます。また、ボタンなども立体的になっています。

A



図A.3 texteditの画面

A.1.6 Xウィンドウマネージャ

Xのもうひとつの重要な要素は**ウィンドウマネージャ**です。ウィンドウマネージャは、ほかのクライアントの面倒をみる特別なXクライアントで、ディスプレイ上でのクライアントウィンドウの配置、ウィンドウの移動やサイズの変更などを管理します。ウィンドウマネージャは、使っているXツールキットによってそれぞれ独自の外観と操作性を持っています。

次の表は、いくつかのウィンドウマネージャをリストアップしたものです。

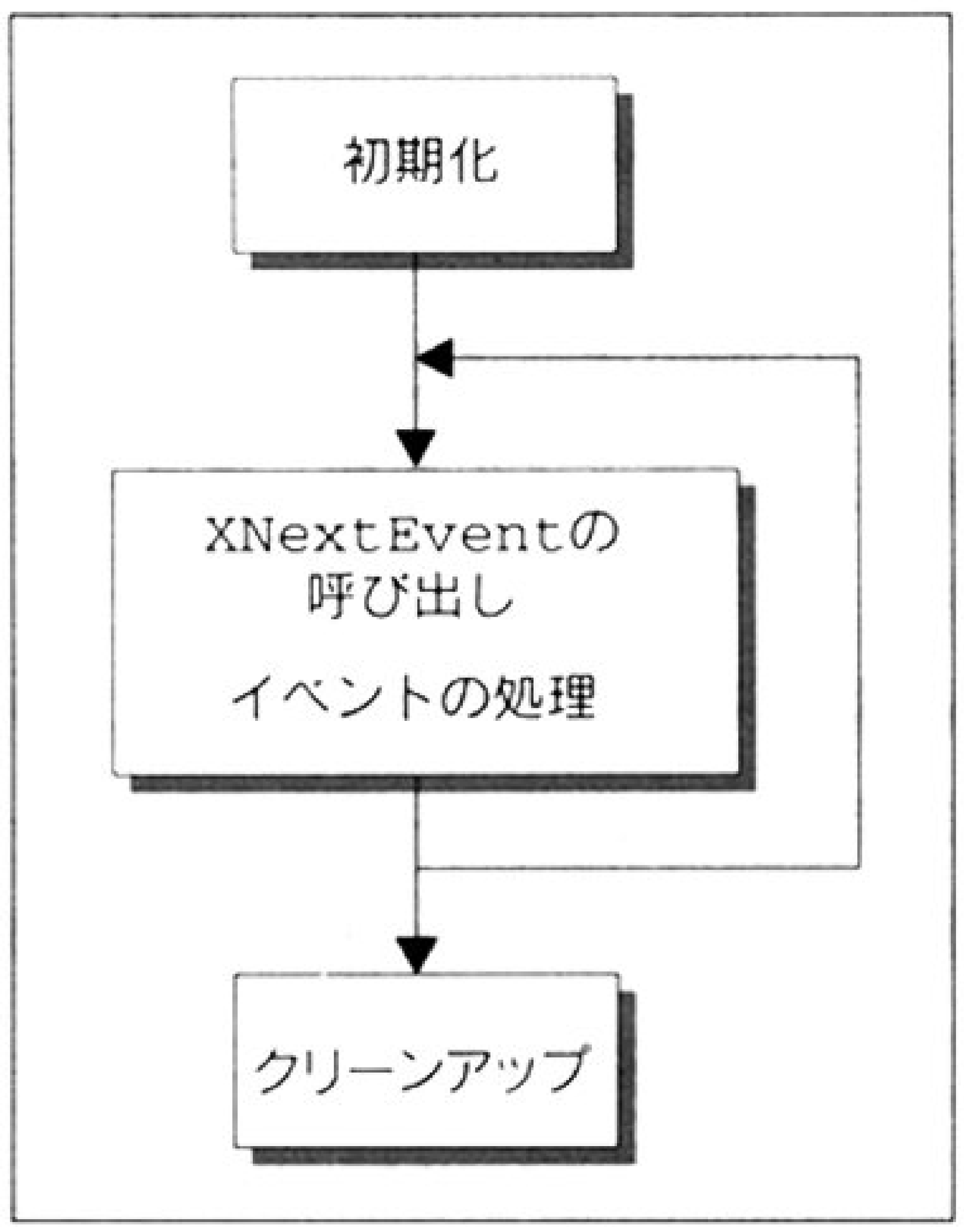
表A.1 代表的なウィンドウマネージャ

ウィンドウマネージャ	説明
`twm`	Tom's（またはTabbed）Window Manager。Xに付属するコンパクトで高速なマネージャ。
`fvwm`	LinuxでよくつかわれているRobert Nation作のウィンドウマネージャ。仮想デスクトップのサポートに加えて、ほかのウィンドウマネージャをエミュレートするための設定ファイルが用意されている。
`fvwm95`	Windows 95のインタフェースをエミュレートする`fvwm`のバージョンのひとつ。
`gwm`	LISPライクなインタープリタでプログラミング可能なGeneric Window Manager。
`olwm`	OpenLookウィンドウマネージャ。
`mwm`	Motifウィンドウマネージャ。

上に示したウィンドウマネージャは、いずれもほとんどのUNIXシステムとLinuxで利用することができます（mwmではラインセンスが必要です）。

A.2 Xのプログラミングモデル

すでに説明したように、X Window Systemではクライアントアプリケーションとxディスプレイサーバーの役割が明確に分離されており、両者はある決められたプロトコルを使って情報をやりとりするようになっています。ここでは、こうしたプログラミング手法に基くXアプリケーションの一般的な構造について、簡単に説明しておきます。



図A.4　Xのプログラミングモデル

A.2.1 スタートアップ

通常、Xアプリケーションは、必要とするリソースの初期化から処理を開始します。まずXディスプレイサーバーとの接続を確立し、使用するカラーとフォントを選択してディスプレイ上にウィンドウを作成します。

クライアントプログラムは、**XOpenDisplay**と**XCloseDisplay**を使ってXサーバーとの接続と切断を行います。

```
Display *XOpenDisplay(chr *display_name);
void XCloseDisplay(Display *display);
```

A

display_nameには接続先のディスプレイを指定します。**display_name**が**NULL**の場合には、環境変数**DISPLAY**の値が使われます。この値は、**hostname:server[.display]**という形式で指定します。この形式からわかるように、1つのホスト上に複数のXサーバーがあってもかまいません。各Xサーバーでは、1つ以上のディスプレイを制御することができます。通常、デフォルトのディスプレイは**:0.0**です。これは、ローカルマシン上で最初に利用可能なサーバーを表します。2番目の画面を指定するには、**:0.1**と指定します。

XOpenDisplayは、選択されたXサーバーに関する情報を含む**Display**構造体を返します。Xサーバーを開くことができなかった場合には、**NULL**を返します。**XOpenDisplay**の呼び出しが成功すれば、クライアントプログラムはXサーバーの使用を開始することができます。

Xサーバーの使用を終えたクライアントプログラムは、**XOpenDisplay**から返された**Display**構造体を指定して**XCloseDisplay**を呼び出さなければなりません。この操作によって、指定されたディスプレイ上でクライアントが作成したすべてのウィンドウとリソースが破棄されます(ただし、**XSetCloseDownMode**を呼び出してシャットダウン時の動作を変更している場合は除きます)。プログラムでは、終了する前に必ず**XcloseDisplay**を呼び出し、未処理のエラーが報告されるようにする必要があります。

ユーザーは、プログラムの動作のほとんどを起動時に指定することができます。多くのXアプリケーションでは、コマンドライン引数、環境変数、設定ファイルを使ってユーザーがアプリケーションをカスタマイズできるようになっています。

たとえば、すでに示したように環境変数**DISPLAY**を使えば、アプリケーションを表示するディスプレイサーバーを指定することができます。このディスプレイサーバーは、ネットワーク上のほかのコンピュータにあってもかまいません。次のコマンドは、**xedit**プログラムを実行し、**alex**という名前のマシンのディスプレイにウィンドウを表示します。

```
$ DISPLAY=alex:0.0 xedit &
```

.Xresources(または場合によって**.Xdefaults**)というファイルを使うと、Xアプリケーションに関するさまざまな設定を行うことができます。各Xアプリケーションは、Xリソースデータベース内の設定エントリを参照します。通常、このXリソースデータベースはXシステムの起動時に作成され、ユーザー独自のローカルな設定を含んでいます。たとえば、ユーザーのホームディレクトリに次のようなエントリを持つ**.Xresources**ファイルがあるとします。

```
xedit*enableBackups: on
```

このエントリによって、**xedit**による編集中のバックアップファイル作成の動作が変わります。こうしたエントリは、一般に次のような形式をしています。

```
Class*Resource: Value
```

また、コマンドラインで次のように指定して**xedit**を起動すると、幅が400ピクセルで高さが200ピクセルの**xedit**のウィンドウが開きます。

```
$ xedit -geometry 400x200
```

ウィンドウサイズの指定方法はプログラムによって異なります。たとえば、**xterm**では行数とカラム数で指定します。したがって、次のコマンドは、50行80桁でターミナルエミュレータを起動することになります。

```
$ xterm -geometry 80x50
```

Xアプリケーションの動作を変更する方法については、それぞれのアプリケーションのマニュアルページを参考にしてください。

A.2.2 メインループ

Xアプリケーションの大部分は、メインループと、イベントに応答するためのコードから構成されています。標準的なXプログラムは、起動後に接続先のXディスプレイサーバーからイベントが送信されてくるのを待ちます。この処理は、ループ内で**XNextEvent**を呼び出すことによって行います。

アプリケーションでは、30以上のイベントを処理する必要があります。イベントの詳しい説明はXプログラミングの専門書にゆずるとして、ここではイベントの種類をいくつか簡単に紹介しておきます。

- キーボードイベント　キーが押された、キーが離されたなど。
- マウスイベント　ボタンが押された、ボタンが離された、マウスが移動した、マウスがウィンドウに入った、マウスがウィンドウから出たなど。
- ウィンドウイベント　ウィンドウが作成された、ウィンドウが破棄された、ウィンドウがフォーカスを取得した、ウィンドウがフォーカスを失った、ウィンドウが表示されたなど。

低水準のXプログラムでは、これらのイベントをはじめ、ほかのイベントにも応答しなければなりません。一方、ツールキットやアプリケーションフレームワークを使うプログラムでは、アプリケーションそのものの構築に注意を集中し、ダイアログボックスなどの洗練されたインタフェースを利用することができます。その場合、低水準のイベントの処理について気にする必要はありません。もちろん、だからといってイベントそのものが存在しなくなるわけではありません。

A.2.3 クリーンアップ

適切に記述されたXプログラムは、実行中に割り当てたXディスプレイリソースを終了時に解放します。プログラムはサーバーとの接続を切断するだけでも終了できますが、この方法では、サーバーが消費するメモリが必要以上に増えてしまうことがあります。プログラムを終了するときは、適切な手順に従うようにしましょう。

付録B

GNU General Public License (GPL)

この付録には、英語版と日本語訳によるGNU General Public License (GPL) の全文を記載します。"copyleft" の考え方を示すこのライセンスは、自由を制限することを目的としている一般のcopyrightとは異なり、逆に自由を拡張しようとしていることが特徴といえます。なお、文章には手を加えていません。

B.1 GNU General Public License

```
          GNU GENERAL PUBLIC LICENSE
             Version 2, June 1991

 Copyright (C) 1989, 1991 Free Software Foundation, Inc.
                     59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

              Preamble

  The licenses for most software are designed to take away your
freedom to share and change it.  By contrast, the GNU General Public
License is intended to guarantee your freedom to share and change free
software--to make sure the software is free for all its users.  This
General Public License applies to most of the Free Software
Foundation's software and to any other program whose authors commit to
using it.  (Some other Free Software Foundation software is covered by
the GNU Library General Public License instead.)  You can apply it to
your programs, too.

  When we speak of free software, we are referring to freedom, not
price.  Our General Public Licenses are designed to make sure that you
have the freedom to distribute copies of free software (and charge for
this service if you wish), that you receive source code or can get it
if you want it, that you can change the software or use pieces of it
in new free programs; and that you know you can do these things.

  To protect your rights, we need to make restrictions that forbid
anyone to deny you these rights or to ask you to surrender the rights.
These restrictions translate to certain responsibilities for you if you
distribute copies of the software, or if you modify it.

  For example, if you distribute copies of such a program, whether
gratis or for a fee, you must give the recipients all the rights that
you have.  You must make sure that they, too, receive or can get the
source code.  And you must show them these terms so they know their
rights.

  We protect your rights with two steps: (1) copyright the software, and
```

(2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1

above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices
stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in
whole or in part contains or is derived from the Program or any
part thereof, to be licensed as a whole at no charge to all third
parties under the terms of this License.

c) If the modified program normally reads commands interactively
when run, you must cause it, when started running for such
interactive use in the most ordinary way, to print or display an
announcement including an appropriate copyright notice and a
notice that there is no warranty (or else, saying that you provide
a warranty) and that users may redistribute the program under
these conditions, and telling the user how to view a copy of this
License. (Exception: if the Program itself is interactive but
does not normally print such an announcement, your work based on
the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If
identifiable sections of that work are not derived from the Program,
and can be reasonably considered independent and separate works in
themselves, then this License, and its terms, do not apply to those
sections when you distribute them as separate works. But when you
distribute the same sections as part of a whole which is a work based
on the Program, the distribution of the whole must be on the terms of
this License, whose permissions for other licensees extend to the
entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest
your rights to work written entirely by you; rather, the intent is to
exercise the right to control the distribution of derivative or
collective works based on the Program.

In addition, mere aggregation of another work not based on the Program
with the Program (or with a work based on the Program) on a volume of
a storage or distribution medium does not bring the other work under
the scope of this License.

3. You may copy and distribute the Program (or a work based on it,
under Section 2) in object code or executable form under the terms of
Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable
source code, which must be distributed under the terms of Sections
1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three
years, to give any third party, for a charge no more than your
cost of physically performing source distribution, a complete
machine-readable copy of the corresponding source code, to be

distributed under the terms of Sections 1 and 2 above on a medium
customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer
to distribute corresponding source code. (This alternative is
allowed only for noncommercial distribution and only if you
received the program in object code or executable form with such
an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for
making modifications to it. For an executable work, complete source
code means all the source code for all modules it contains, plus any
associated interface definition files, plus the scripts used to
control compilation and installation of the executable. However, as a
special exception, the source code distributed need not include
anything that is normally distributed (in either source or binary
form) with the major components (compiler, kernel, and so on) of the
operating system on which the executable runs, unless that component
itself accompanies the executable.

If distribution of executable or object code is made by offering
access to copy from a designated place, then offering equivalent
access to copy the source code from the same place counts as
distribution of the source code, even though third parties are not
compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program
except as expressly provided under this License. Any attempt
otherwise to copy, modify, sublicense or distribute the Program is
void, and will automatically terminate your rights under this License.
However, parties who have received copies, or rights, from you under
this License will not have their licenses terminated so long as such
parties remain in full compliance.

5. You are not required to accept this License, since you have not
signed it. However, nothing else grants you permission to modify or
distribute the Program or its derivative works. These actions are
prohibited by law if you do not accept this License. Therefore, by
modifying or distributing the Program (or any work based on the
Program), you indicate your acceptance of this License to do so, and
all its terms and conditions for copying, distributing or modifying
the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the
Program), the recipient automatically receives a license from the
original licensor to copy, distribute or modify the Program subject to
these terms and conditions. You may not impose any further
restrictions on the recipients' exercise of the rights granted herein.
You are not responsible for enforcing compliance by third parties to
this License.

7. If, as a consequence of a court judgment or allegation of patent
infringement or for any other reason (not limited to patent issues),

conditions are imposed on you (whether by court order, agreement or
otherwise) that contradict the conditions of this License, they do not
excuse you from the conditions of this License. If you cannot
distribute so as to satisfy simultaneously your obligations under this
License and any other pertinent obligations, then as a consequence you
may not distribute the Program at all. For example, if a patent
license would not permit royalty-free redistribution of the Program by
all those who receive copies directly or indirectly through you, then
the only way you could satisfy both it and this License would be to
refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under
any particular circumstance, the balance of the section is intended to
apply and the section as a whole is intended to apply in other
circumstances.

It is not the purpose of this section to induce you to infringe any
patents or other property right claims or to contest validity of any
such claims; this section has the sole purpose of protecting the
integrity of the free software distribution system, which is
implemented by public license practices. Many people have made
generous contributions to the wide range of software distributed
through that system in reliance on consistent application of that
system; it is up to the author/donor to decide if he or she is willing
to distribute software through any other system and a licensee cannot
impose that choice.

This section is intended to make thoroughly clear what is believed to
be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in
certain countries either by patents or by copyrighted interfaces, the
original copyright holder who places the Program under this License
may add an explicit geographical distribution limitation excluding
those countries, so that distribution is permitted only in or among
countries not thus excluded. In such case, this License incorporates
the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions
of the General Public License from time to time. Such new versions will
be similar in spirit to the present version, but may differ in detail to
address new problems or concerns.

Each version is given a distinguishing version number. If the Program
specifies a version number of this License which applies to it and "any
later version", you have the option of following the terms and conditions
either of that version or of any later version published by the Free
Software Foundation. If the Program does not specify a version number of
this License, you may choose any version ever published by the Free Software
Foundation.

10. If you wish to incorporate parts of the Program into other free
programs whose distribution conditions are different, write to the author

```
to ask for permission.  For software which is copyrighted by the Free
Software Foundation, write to the Free Software Foundation; we sometimes
make exceptions for this.  Our decision will be guided by the two goals
of preserving the free status of all derivatives of our free software and
of promoting the sharing and reuse of software generally.

                            NO WARRANTY

  11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY
FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW.  EXCEPT WHEN
OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES
PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED
OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.  THE ENTIRE RISK AS
TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU.  SHOULD THE
PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING,
REPAIR OR CORRECTION.

  12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING
WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR
REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES,
INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING
OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED
TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY
YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER
PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE
POSSIBILITY OF SUCH DAMAGES.

                     END OF TERMS AND CONDITIONS

            How to Apply These Terms to Your New Programs

  If you develop a new program, and you want it to be of the greatest
possible use to the public, the best way to achieve this is to make it
free software which everyone can redistribute and change under these terms.

  To do so, attach the following notices to the program.  It is safest
to attach them to the start of each source file to most effectively
convey the exclusion of warranty; and each file should have at least
the "copyright" line and a pointer to where the full notice is found.

    <one line to give the program's name and a brief idea of what it does.>
    Copyright (C) 19yy  <name of author>

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.
```

```
    You should have received a copy of the GNU General Public License
    along with this program; if not, write to the Free Software
    Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
    Gnomovision version 69, Copyright (C) 19yy name of author
    Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w'
    This is free software, and you are welcome to redistribute it
    under certain conditions; type `show c' for details.
```

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w' and `show c'; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
  Yoyodyne, Inc., hereby disclaims all copyright interest in the program
  `Gnomovision' (which makes passes at compilers) written by James Hacker.

  <signature of Ty Coon>, 1 April 1989
  Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

B.2 GNU一般公有使用許諾書

GNU一般公有使用許諾書
＝＝＝＝＝＝＝＝＝＝＝＝

1991年6月，バージョン2

---

* 【注意】 現在、このバージョン2の発行者(FSF)住所は、正式に新しい住所の 59 Temple Place - Suite 330, Boston, MA 02111-1307, USAに変わっている。

何人も、以下の内容を変更しないでそのまま複写する場合に限り、本使用許諾書を複製したり頒布することができます。

はじめに

ほとんどのソフトウェアの使用許諾は、ソフトウェアを共有し、変更するユーザの自由を奪うことを意図しています。それに対して、我々のGNU一般公有使用許諾は、フリー・ソフトウェアを共有したり変更する自由をユーザに保証するためのもの、即ちフリー・ソフトウェアがそのユーザ全てにとってフリーであることを保証するためのものです。本使用許諾は、Free Software Foundationのほとんど全てのソフトウェアに適用されるだけでなく、プログラムの作成者が本使用許諾に依るとした場合のそのプログラムにも適用することができます。(その他の Free Software Foundation のソフトウェアのいくつかは、本許諾書ではなく、GNUライブラリ一般公有使用許諾で保護されます。)あなたは自分のプログラムにもこれを適用できます。我々がフリー・ソフトウェアについて言う場合は自由のことに言及しているのであって、価格のことではありません。我々の一般公有使用許諾の各条項は、次の事柄を確実に実現することを目的として立案されています。

・ フリー・ソフトウェアの複製物を自由に頒布できること(そして、望むならあなたのこのサービスに対して対価を請求できること)。

・ ソース・コードを実際に受け取るか、あるいは、希望しさえすればそれを入手することが可能であること。

・ 入手したソフトウェアを変更したり、新しいフリー・プログラムの一部として使用できること。

・ 以上の各内容を行なうことができるということをユーザ自身が知っていること。

このようなユーザの権利を守るために、我々は、何人もこれらの権利を否定したり、あるいは放棄するようにユーザに求めることはできないという制限条項を設ける必要があります。これらの制限条項は、ユーザが、フリー・ソフトウェアの複製物を頒布したり変更しようとする場合には、そのユーザ自身が守るべき義務ともなります。例えば、あなたがフリー・ソフトウェアの複製物を頒布する場合、有償か無償かにかかわらず、あなたは自分の持っている権利を全て相手に与えなければなりません。あなたは、相手もまたソース・コードを受け取ったり入手できるということを認めなければなりません。さらにあなたは、彼らが自分たちの権利を知るように、これらの条項を知らしめなければなりません。

我々は次の２つの方法でユーザの権利を守ります。（１）ソフトウェアに著作権を主張し、（２）本使用許諾の条項の下でソフトウェアを複製・頒布・変更する権利をユーザに与えます。

また、各作成者や我々自身を守るために、本フリー・ソフトウェアが無保証であることを全ての人々が了解している必要があります。さらに、他の誰かによって変更されたソフトウェアが頒布された場合、受領者はそのソフトウェアがオリジナル・バージョンではないということを知らされる必要があります。それは、他人の関与によって原開発者に対する評価が影響されないようにするためです。

最後に、どのフリー・プログラムもソフトウェア特許に絶えず脅かされています。我々は、フリー・プログラムの再頒布者が個人的に特許権を取得し、事実上そのプログラムを自分の財産にしてしまうという危険を避けたいと願っています。これを防ぐために我々は、いずれの特許も、誰でも自由に使用できるように使用許諾されるべきか、あるいは何人に対しても全く使用させないかの、いずれかにすべきであることを明らかにしてきました。

複写・頒布・変更に対する正確な条項と条件を次に示します。

ＧＮＵ一般公有使用許諾の下での複製、頒布、変更に関する条項と条件
＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝

１． 本使用許諾は、本一般公有使用許諾の各条項に従って頒布されるという著作権者からの告知文が表示されているプログラムやその他の作成物に適用されます。以下において「プログラム」とは、そのようなプログラムや作成物を指すものとし、また、「プログラム生成物」とは、上述した「プログラム」自身、または、著作権法下における全ての派生物；すなわち、その「プログラム」の全部又は一部を、そのまま又は変更して、且つ／又は他の言語に変換して、内部に組み込んだ作成物を意味します。（以下、言語変換は「変更」という用語の中に無条件に含まれるものとします。）本使用許諾によって許諾を受ける者を「あなた」と呼びます。

複製、頒布、変更以外の行為は本使用許諾の対象としません。それらは本使用許諾の範囲外です。「プログラム」を実行させる行為に関して制約はありません。「プログラム」の出力は、（ 「プログラム」を実行させて作成させたかどうかとは無関係に） その内容が「プログラム生成物」である場合に限り本使用許諾の対象となります。これが当てはまるかどうかは、「プログラム」が何をするものかに依ります。

２． あなたは、どのような媒体上へ複製しようとする場合であっても、入手した「プログラム」のソース・コードをそのままの内容で複写した上で適正な著作権表示と保証の放棄を明確、且つ適正に付記する場合に限り、複製又は頒布することができます。その場合、本使用許諾及び無保証に関する記載部分は、全て元のままの形で表示してください。また、「プログラム」の頒布先に対しては、「プログラム」と共に本使用許諾書の写しを渡してください。複製物の引き渡しに要する実費は請求することができます。また、あなた独自の保証を行なう場合はそれを有償とすることができます。

３． 次の各条件を全て満たしている限り、あなたは、「プログラム」又はその一部分を変更して「プログラム生成物」とすることができ、さらに、変更版や右作成物を上記第２項に従って複製又は頒布することもできます。

（ａ）ファイルを変更した旨とその変更日とを、変更したファイル上に明確に表示すること。

（ｂ）変更したか否かを問わず、凡そ「プログラム」又はその一部分を内部に組み込んでいるか又はそれから派生した生成物を頒布

する場合には、その全体を本使用許諾の条項に従って第三者へ無償で使用許諾すること。

（c）変更したプログラムが実行時に通常の対話的な方法でコマンドを読むようになっているとすれば、最も普通の方法で対話的にそのプログラムを実行する時に、次の内容を示す文言がプリンタへ印字されるか、或いは画面に表示されること。

・適切な著作権表示。
・無保証であること（あなたが独自に保証する場合は、その旨)。
・頒布を受ける者も、本使用許諾と同一の条項に従って「プログラム」を再頒布できること。
・頒布を受ける者が本使用許諾書の写しを参照する方法。

（例外として、「プログラム」自体は対話的であっても起動時の文言を通常は印字しないのならば、あなたの「プログラム生成物」はこのような文言を印字する必要はありません。）

これらの要件は変更された作成物にも全て適用されます。その変更版の或る部分が「プログラム」の派生物ではなく、しかもそれ自体独立で異なる作成物だと合理的に考えられる場合、あなたがそれらを別の作成物として頒布した時は、本使用許諾とその条項はそれらの部分には適用されません。しかし、それらを「プログラム生成物」の一部として頒布する場合は、全体が本使用許諾の条項に従って頒布されなければならず、使用許諾を受ける他の全ての者に対する許諾もプログラム全体にわたって与えられなければならず、結果として、誰が書いたかにかかわらず、全ての部分に本使用許諾が適用されなければなりません。

このように、本条項の意図するところは、完全にあなたによって書かれた作成物について、権利を要求したり、あなたと権利関係を争うことではありません。むしろその目的は、作成物が「プログラム生成物」である場合にその派生物や集合物の頒布を規制することにあります。

さらに、「プログラム」（又は「プログラム生成物」） と「プログラム生成物」とはならない他のプログラムとを、単に保管や頒布のために同一の媒体上にまとめて記録したとしても、本使用許諾は他のプログラムには適用されません。

4．あなたは、以下のうちいずれか1つを満たす限り、上記第2項及び第3項に従って「プログラム」（又は、上記第3項で言及している「プログラム生成物」）をオブジェクト・コード又は実行可能な形式で複

製及び頒布することができます。

（a）対応する機械読み取り可能なソース・コード一式を一緒に引き渡すこと。その場合、そのソース・コードの引き渡しは上記第2項及び第3項に従って、通常ソフトウェアの交換に用いられる媒体で行なわれること。

（b）少なくとも3年間の有効期間を定め、且つその期間内であれば対応する機械読み取り可能なソース・コード一式の複製を、ソース頒布に関わる実費以上の対価を要求せずに提供する旨、及びその場合には上記第2項及び第3項に従って、通常ソフトウェアの交換に用いられる媒体で提供される旨を記載した書面を、第三者に一緒に引き渡すこと。

（c）対応するソース・コード頒布の申し出に際して、あなたが得た情報を一緒に引き渡すこと。（この選択肢は、営利を目的としない頒布であって、且つあなたが上記の（b）項に基づいて、オブジェクト・コード或いは実行可能形式のプログラムしか入手していない場合に限り適用される選択項目です。）

なお、ソース・コードとは、変更作業に適した記述形式を指します。また、実行可能形式のファイルに対応するソース・コード一式とは、それに含まれる全モジュールに対応する全てのソース・コード、及びあらゆる関連のインタフェース定義ファイル、及び実行を可能にするコンパイルとインストールの制御に関する記述を指します。特別な例外として、実行可能なファイルが動作するオペレーティング・システムの主要な構成要素（コンパイラ、カーネルなど） と共に（ソース・コード又はバイナリのどちらかで） 頒布されているものについては、その構成要素自体が実行形式に付随していない場合に限り、頒布されるソース・コードに含める必要はありません。

実行可能形式またはオブジェクト・コードの頒布が、指示された場所からの複製のためのアクセス権の賦与である場合、同じ場所からのソース・コードの複製のための同等なアクセス権を賦与すれば、たとえ第三者にオブジェクト・コードと共にソースの複製を強いなくとも、ソース・コードを頒布したものとみなします。

5．本使用許諾が明示的に許諾している場合を除き、あなたは、「プログラム」を複製、変更、サブライセンス、頒布することができません。本使用許諾に従わずに「プログラム」を複製、変更、サブライセンス、頒布しようとする行為は、それ自体が無効であり、且つ、本使用許諾があなたに許諾している「プログラム」の権利を自動的に消滅させま

す。その場合、本使用許諾に従ってあなたから複製物やその権利を得ている第三者は、本使用許諾に完全に従っている場合に限り、引続き有効な使用権限を持つものとします。

6. あなたはまだ同意の印として署名していないので、本使用許諾を受け入れる必要はありません。しかし、あなたに「プログラム」又はその派生物を変更又は再頒布する許可を与えるものは本使用許諾以外にはありません。これらの行為は、あなたがもし本使用許諾を受け入れないのであれば、法律によって禁じられます。従って、あなたが「プログラム」（又は「プログラム生成物」）の変更又は頒布を行えば、それ自体であなたは本使用許諾を受け入れ、且つ、「プログラム」又はその「プログラム生成物」の複製、頒布、変更に関するこれらの条項と条件の全てを受け入れたことを示します。

7. あなたが「プログラム」（又はその「プログラム生成物」）を再頒布すると自動的に、その受領者は、元の使用許諾者から、本使用許諾の条項に従って「プログラム」を複製、頒布、変更することを内容とする使用許諾を受けたものとします。あなたは、受領者に許諾された権利の行使について、さらに制約を加えることはできません。あなたには、第三者に本使用許諾の受け入れを強いる責任はありません。

8. 裁判所の判決、又は特許侵害の申し立て、又は（特許問題に限らない）何らかの理由の結果として、あなたに課せられた条件が本使用許諾と相入れないものであったとしても（裁判所の命令、契約、その他によるものであれ）、本使用許諾の条件が免除されるものではありません。本使用許諾による責務と、その他の何らかの関連責務を同時に満たす態様で頒布することができないならば、あなたは「プログラム」を全く頒布してはいけません。例えば、特許権の内容が、あなたから直接又は間接に複製を受け取った全ての人に使用料のないプログラムの再頒布を許さないものであれば、あなたがかかる特許上の要請と本使用許諾の両方を満足させる方法は、「プログラム」の頒布を完全に断念することだけです。

本条項の或る部分が何らかの特別な状況下で無効または適用不可能になった場合、本条項のその他の残りの部分が適用されるように意図されており、また、本条項は全体としてその他の状況に当てはまるように意図されています。

本条項の目的は、特許やその他の財産権を侵害したり、そのような権利に基づく主張の妥当性を争うようにあなたに勧めることではありません。本条項の唯一の目的は、フリー・ソフトウェアの頒布システムの完全性を守ることで、それは公有使用許諾の実践によって履行され

ます。多くの人々が、このシステムの一貫した適用を信頼して、このシステムを通じて頒布されている幅広い範囲のソフトウェアに惜しみない貢献をしてくれました。作成者や寄贈者が他の何らかのシステムを通じてソフトウェアを頒布したいと決めることは彼らの自由意志であり、使用許諾を受ける者はその選択を強いることはできません。

本条項は、本使用許諾の他の条項の意味内容が何であるかを完全に明らかにすることを意図しています。

9．「プログラム」の頒布・使用が、ある国において特許又は著作権で保護されたインタフェースのどちらかで制限される場合、「プログラム」を本使用許諾下においた原著作権保持者は、その国を除外する旨の明示的な頒布地域制限を加え、それ以外の（除外されない） 国に限定して頒布が許されるようにすることができます。そのような場合、その制限を本使用許諾の本文にあたかも書かれているかのように本使用許諾の中に組み入れられるものとします。

10．Free Software Foundation は随時、本一般公有使用許諾の改訂版、又は新版を公表することがあります。そのような新しいバージョンは、現行のバージョンと基本的に変わるところはありませんが、新しい問題や懸案事項に対応するために細部では異なるかもしれません。

各バージョンは、バージョン番号によって区別します。「プログラム」中に本使用許諾のバージョン番号の指定がある場合は、その指定されたバージョンか、又はその後に Free Software Foundation から公表されているいずれかのバージョンから１つを選択して、その条項と条件に従ってください。「プログラム」中に本使用許諾のバージョン番号の指定がない場合は、Free Software Foundation が公表したどのバージョンでも選択することができます。

11．「プログラム」の一部を頒布条件の異なる他のフリー・プログラムに組み込みたい場合は、その開発者に書面で許可を求めてください。Free Software Foundation が著作権を持っているソフトウェアについては、 Free Software Foundation へ書面を提出してください。このような場合に対応するために我々は例外的処理をすることもありますが、その判断基準となるのは、次の２つの目標の実現に合致するか否かという点です。即ち、１つは我々のフリー・ソフトウェアの全ての派生物をフリーな状態に保つことであり、もう１つはソフトウェアの共有と再利用とを広く促進させることです。

無保証

１２．「プログラム」は無償で使用許諾されますので、適用法令の範囲内で、「プログラム」の保証は一切ありません。著作権者やその他の第三者は全く無保証で「そのまま」の状態で、且つ、明示か暗黙であるかを問わず一切の保証をつけないで提供するものとします。ここでいう保証とは、市場性や特定目的適合性についての暗黙の保証も含まれますが、それに限定されるものではありません。「プログラム」の品質や性能に関する全てのリスクはあなたが負うものとします。「プログラム」に欠陥があるとわかった場合、それに伴う一切の派生費用や修理・訂正に要する費用は全てあなたの負担とします。

１３．適用法令の定め、又は書面による合意がある場合を除き、著作権者や上記許諾を受けて「プログラム」の変更・再頒布を為し得る第三者は、「プログラム」を使用したこと、または使用できないことに起因する一切の損害について何らの責任も負いません。著作権者や前記の第三者が、そのような損害の発生する可能性について知らされていた場合でも同様です。なお、ここでいう損害には通常損害、特別損害、偶発損害、間接損害が含まれます（データの消失、又はその正確さの喪失、あなたや第三者が被った損失、他のプログラムとのインタフェースの不適合化、等も含まれますが、これに限定されるものではありません）。

以上

注意

＊＊

英文文書 (GNU General Public License) を正式文書とする。この和文文書は弁護士の意見を採り入れて、できるだけ正確に英文文書を翻訳したものであるが、法律的に有効な契約書ではない。

和文文書自体の再配布に関して

＊＊＊＊＊＊＊＊＊＊＊＊＊＊

いかなる媒体でも次の条件がすべて満たされている場合に限り、本和文文書をそのまま複写し配布することを許可する。また、あなたは第三者に対して本許可告知と同一の許可を与える場合に限り、再配布することが許可されています。

・受領、配布されたコピーに著作権表示および本許諾告知が前もって載せられていること。

・コピーの受領者がさらに再配布する場合、その配布者が本告知と同じ許可を与えていること。

・和文文書の本文を改変しないこと。

あなたの新しいプログラムにこれらの条項を適用する方法
＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝

あなたが新しくプログラムを作成し、それを公用に供したい場合は、プログラムをフリー・ソフトウェアにして、全ての人々が以上の各条項に従ってこれを再頒布や変更をすることができるようにするのが最良の方法です。

そうするためには、プログラムに以下の表示をしてください。その場合、無保証であるということを最も効果的に伝えるために、ソース・ファイルの冒頭にその全文を表示すれば最も安全ですが、その他の方法で表示する場合でも、「著作権表示」と全文を読み出す為のアドレスへのポインタだけはファイル上に表示しておいてください。

＜プログラム名とどんな動作をするものかについての簡単な説明の行＞
Ｃｏｐｙｒｉｇｈｔ（Ｃ） １９○○年、＜著作権者名＞

本プログラムはフリー・ソフトウェアです。あなたは、Free Software Foundation が公表したGNU 一般公有使用許諾の「バージョン２」或いはそれ以降の各バージョンの中からいずれかを選択し、そのバージョンが定める条項に従って本プログラムを再頒布または変更することができます。

本プログラムは有用とは思いますが、頒布にあたっては、市場性及び特定目的適合性についての暗黙の保証を含めて、いかなる保証も行ないません。詳細についてはGNU 一般公有使用許諾書をお読みください。

あなたは、本プログラムと一緒にGNU 一般公有使用許諾の写しを受け取っているはずです。そうでない場合は、Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA＊へ手紙を書いてください。

---

* 【注意】 現在、このバージョン2の発行者(FSF)住所は、正式に新しい住所の 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA に変わっている。

また、ユーザが電子メイルや書信であなたと連絡をとる方法についての情報も書き添えてください。

プログラムが対話的に動作する場合は、対話モードで起動した時に次のような短い告知文が表示されるようにしてください。

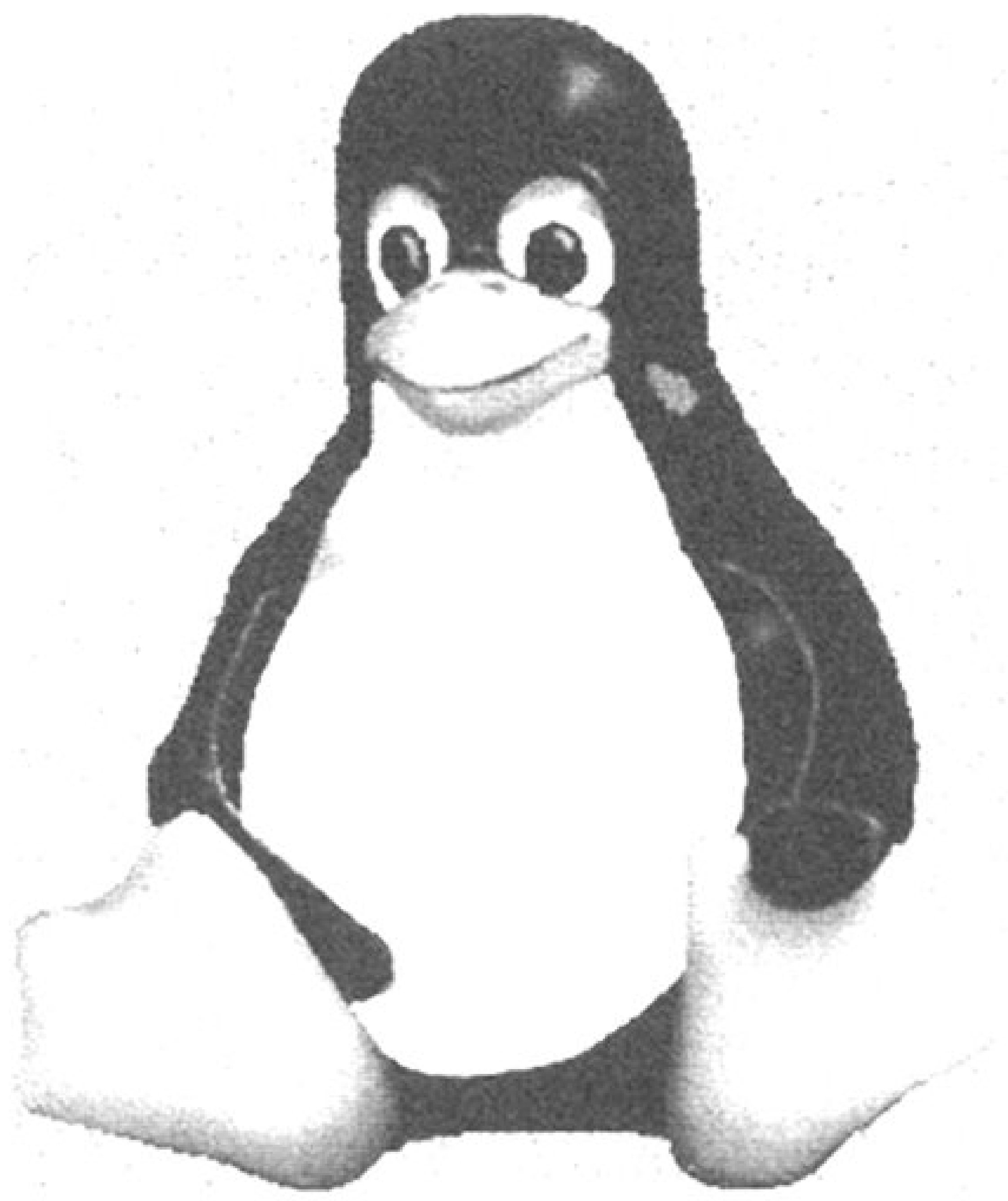
> Ｇｎｏｍｏｖｉｓｉｏｎ バージョン６９、Ｃｏｐｙｒｉｇｈｔ（Ｃ）１９○○年 <著作権者名>
>
> Gnomovision は完全に無保証です。詳細は show w とタイプしてください。これはフリー・ソフトウェアなので、特定の条件の下でこれを再頒布することができます。詳細は show c とタイプしてください。

上記のshow w やshow c は各々、本一般公有使用許諾の関連する部分を表示するコマンドを指します。もちろん、あなたが使うこれらのコマンドはshow w やshow c といった呼び名でなくても構いません。さらに、それらのコマンドはあなたのプログラムに合わせる為に、マウスでクリックしたりメニュー形式にすることもできます。

また、必要と認めた場合には、あなたの雇い主(あなたがプログラマとして働いている場合) や在籍する学校から、そのプログラムに対する「著作権放棄」を認めた署名入りの書面を入手してください。ここにその文例を載せます。名前は変えてください。

> Yoyodyne, Inc. は、James Hacker が開発したプログラム `Gnomovision' (コンパイラにつなげるプログラム) についての著作権法上の全ての権利を放棄する。
>
> <Ｔｙ Ｃｏｏｎ の署名>, １ Ａｐｒｉｌ １９８９
> Ｔｙ Ｃｏｏｎ, 副社長

本一般公有使用許諾は、あなたのプログラムを財産権の対象となっている他のプログラムに組み込むことは認めていません。あなたのプログラムがサブルーチン・ライブラリであって、あなたがそのライブラリを財産権の対象となっている他のアプリケーションとリンクさせることによって、さらに有用なものにしようとする場合には、本使用許諾書の代わりに、GNU ライブラリ一般公有使用許諾書に従ってください。



付録C

参考文献

Advanced Programming in the UNIX Environment
W. Richard Stevens
Addison-Wesley, 1992
ISBN 0-201-56317-7
大木敦雄 訳『詳解UNIXプログラミング』
ソフトバンク 1994
ISBN 4-89052-524-6

Advanced UNIX Programming
Marc J. Rochkind
Prentice Hall, 1986
ISBN 0-13-011800-1
福崎俊弘 訳『UNIXシステムコール・プログラミング』
アスキー 1989
ISBN 4-87148-260-X

The UNIX Programming Environment
Brian W. Kernighan & Rob Pike
Prentice Hall, 1984
ISBN 0-13-937681-X
石田晴久 監訳『UNIXプログラミング環境』
アスキー 1985
ISBN 4-87148-351-7

The Design of the UNIX Operating System
Maurice J. Bach
Prentice Hall, 1986
ISBN 0-13-201799-7

POSIX Programmer's Guide
Donald Lewine
O'Reilly & Associates, 1991
ISBN 0-937175-73-0

Programming with GNU Software
Mike Loukides & Andy Oram
O'Reilly & Associates, 1996
ISBN 1-56592-112-7
引地美恵子 引地信之 共訳『GNUソフトウェアプログラミング』
オーム社 1999

ISBN 4-900900-20-6

Linux Application Development
Michael K. Johnson & Erik W. Troan
Addison-Wesley, 1998
ISBN 0-201-30821-5
株式会社クイック 訳『プログラミングLinux』
アスキー 1998
ISBN 4-7561-2044-X

Linux Kernel Internals
Michael Beck, Harald Bohme, Mirko Dziadzka, Ulrich Kunitz, Robert Magnus & Dirk Verworner
Addison-Wesley 1997
ISBN 0-201-33143-8

Learning the bash Shell
Cameron Newham & Bill Rosenblatt
O'Reilly & Associates, 1998
ISBN 1-56592-347-2
QUIPU LLC 遠藤美代子 訳『入門bash 第2版』
オーム社 1998
ISBN 4-900900-78-8

Termcap and Terminfo
John Strang, Tim O'Reilly & Linda Mui
O'Reilly & Associates, 1991
ISBN 0-937175-22-6
鈴木麗 監訳『termcap & terminfo』
アスキー 1992
ISBN 4-7561-0271-9

Programming with curses
John Strang
O'Reilly & Associates, 1986
ISBN 0-937175-02-1

Managing Projects with make
Andrew Oram & Steve Talbott
O'Reilly & Associates, 1991

ISBN 0-937175-90-0
矢吹道郎 監訳／菊池 彰 訳『make改訂版』
オーム社 1997
ISBN 4-900900-60-5

Unix Network Programming
W. Richard Stevens
Prentice Hall, 1990
ISBN 0-13-949876-1
篠田陽一 訳『UNIXネットワークプログラミング』
トッパン 1992
ISBN 4-8101-8509-5

Unix Network Programming, Volume 1: Networking APIs
W. Richard Stevens
Prentice Hall, 1997
ISBN 0-13-490012-X

Unix Network Programming, Volume 2: Interprocess Communications
W. Richard Stevens
Prentice Hall, 1998
ISBN 0-13-081081-9

X Window Systems Programming and Applications with Xt
Douglas A. Young
Prentice-Hall, 1989
ISBN 0-13-972167-3
川手恭輔 訳『X Toolkitプログラミング』
トッパン 1990
ISBN 4-8101-8502-8

Linux Device Drivers
Alessandro Rubini
O'Reilly & Associates, 1998
ISBN 1-56592-292-1
山崎康宏 山崎邦子 共訳『LINUXデバイスドライバ』
オーム社 1998
ISBN 4-900900-73-7

索引

記号

A

B

E

F

G

N

O

P

Q

R

S

Z

あ

か

さ

た

な

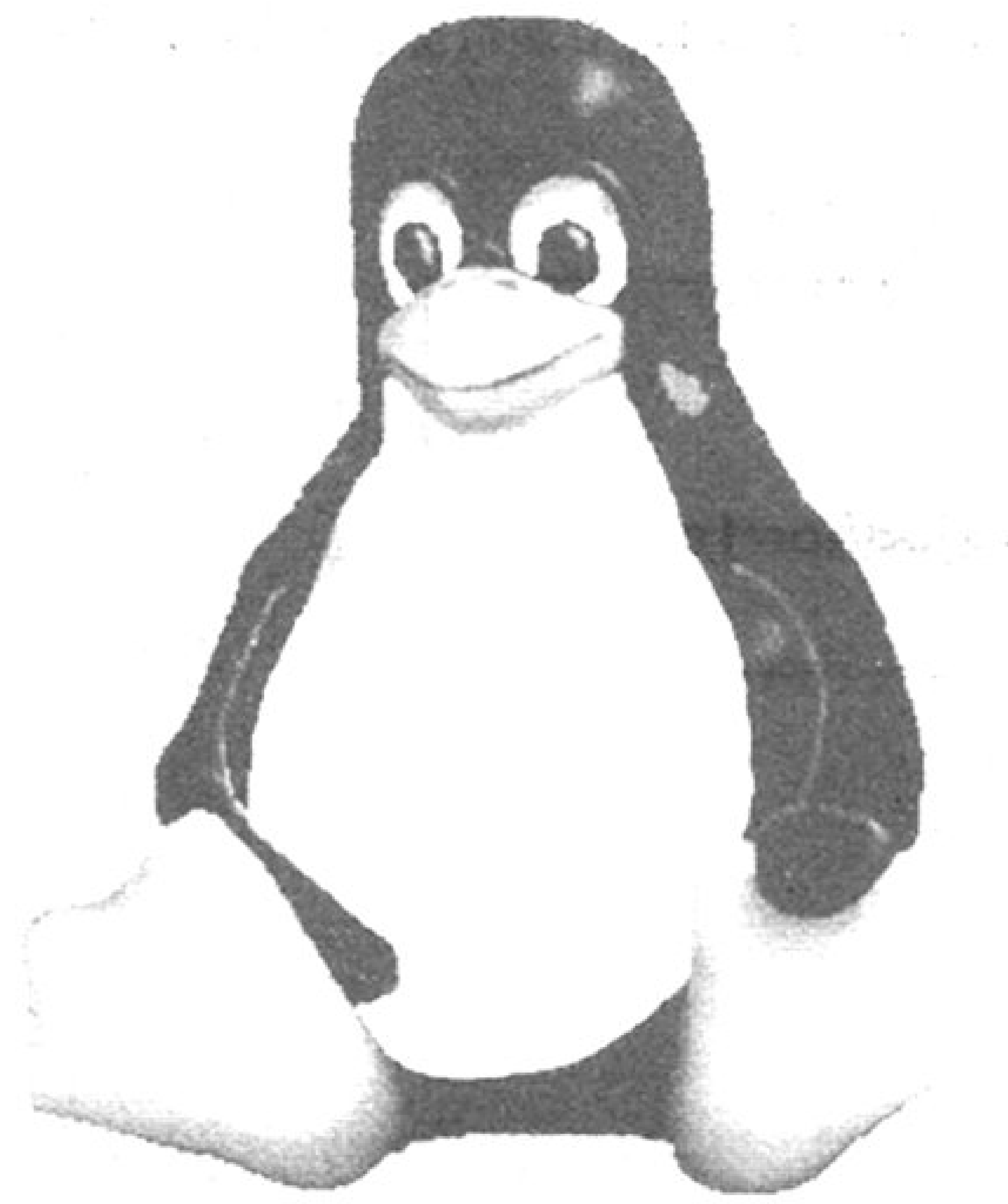
は

ま

や

ら

わ



●本書の内容に関するご質問について

本書の内容に関するご質問は、封書またはファクスでお送りください。封書の場合は、返信用封筒と切手を同封のうえ、「〒103-8501 東京都中央区日本橋箱崎町24-1 ソフトバンク株式会社 出版事業部 書籍出版局 第1書籍編集部」までお願いいたします。ファクスの場合は、書籍名、返信ファクス番号を明記のうえ、ファクス番号03-5642-7605にお送りください。また、本書の内容以外のご質問、および電話でのお問い合わせには応じかねますのでご了承ください。

本書のサポートページ：

http://www.softbank.co.jp/books/editor/1st/linux/

Linux プログラミング

1999年3月27日 初版発行

著者 Neil Matthew, Richard Stones（ニール マシュー、リチャード ストーンズ）

訳者 葛西重夫（かさいしげお）

制作 風工舎

発行者 岡崎 眞

発行所 ソフトバンク株式会社 出版事業部（1999年3月まで）
ソフトバンク パブリッシング株式会社（1999年4月1日から）
〒103-8501 東京都中央区日本橋箱崎町24-1
販売 03-5642-8101
編集 03-5642-8144

印刷 文唱堂印刷株式会社

装丁 早川いくを

落丁本、乱丁本は小社販売局にてお取り替えします。
定価はカバーに記載されております。

Printed in Japan ISBN4-7973-0819-2

SOFT BANK Publishing

wrox

ISBN4-7973-0819-2

C0055 ¥4200E

9784797308198

定価 本体4,200円 ＋税

1920055042000

"The most important design issue...
is the fact that Linux is supposed to be fun..."

Linus Torvalds